Efficient Security and Privacy Enhancing Solutions in Untrusted Environments

Seungyeop Han

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2016

Reading Committee:

Thomas E. Anderson, Chair

Arvind Krishnamurthy, Chair

Franziska Roesner

Program Authorized to Offer Degree:
Computer Science and Engineering

University of Washington

Abstract

Efficient Security and Privacy Enhancing Solutions
in Untrusted Environments

Seungyeop Han

Co-Chairs of the Supervisory Committee:
Professor Thomas E. Anderson
UW Computer Science & Engineering

Associate Professor Arvind Krishnamurthy
UW Computer Science & Engineering

Today, it is common to connect to Internet-based services through a variety of devices. While using the Internet, a user's personal information is exposed to untrusted or unreliable environments, from the applications they are using, to the networks delivering packets, to cloud-based remote services. As personal information increases in value, the incentives for remote services to collect as much of it as possible increase as well. On the other hand, users do not have much control over information exposure, while the risk is high as it is irreversible once it occurs.

Despite the increasing security and privacy risk and much attention from research community and developers, many privacy issues remain unsolved. This dissertation explores the answers to the question: *Can we design security and privacy enhancing systems in the current untrusted environment?* In answering the question, my dissertation considers and tackles two key challenges—untrusted cloud services and linkability of user behavior by providing users with control over how and which of their information is exposed to other parties. It presents the solutions with two systems: MetaSync, a secure and reliable file synchronization service across multiple untrusted service providers, and the IPv6 pseudonym abstraction, a cross-layer architecture allowing users to have flexible control of linkability.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

This dissertation would not have been possible without the support and collaboration of many people.

I have been fortunate to have three great advisors, Thomas Anderson, Arvind Krishnamurthy, and David Wetherall. Throughout six years of my study, I could get all advices and guidance whenever I needed from them, and I learned a lot to become an independent researcher. I would like to thank Matthai Philipose and Jaeyeon Jung who have guided many parts of my research and served as my mentors in addition to my advisors. I am also grateful to other members of my dissertation committee, Franzi Roesner and Sreeram Kannan, for their feedbacks and supports. Looking back, my long journey of studying computer science started from KAIST, and I am especially grateful to Sue Moon for her continued mentorship.

I have enjoyed collaboration with friends and colleagues in and out of UW CSE. I would like to thank Vincent Liu; it was great to start Ph.D. at the same time with him from the same lab. I thank Haichen Shen; I collaborate with him for many research projects and I have always appreciated his diligence. I thank Taesoo Kim; MetaSync started from his idea and he always inspires me. It has been great to be in UW CSE and in the Networks Lab. I could find not only many brilliant coworkers but also good friends with whom I can share much time. I thank all the past and current members but name a few of them: Will, Ray, Qiao, Danyang, Sophia, Simon, Ravi, Qifan, and Colin. I thank Melody Kadenko for her great supports for all administrative things, and thank Lindsay Michimoto and Elise DeGoede for grad school guidance.

Much of my research which includes the work in this dissertation has been sup-

ported in part by KFAS (Korea Foundation for Advanced Studies) fellowship, the National Science Foundation and the Defense Advanced Research Projects Agency.

Lastly, I thank my parents, Sangtae Han and Sunheuy Back, and my brother, Bongyeop, for their unconditional love and supports. And a special thanks to my fiancé, Yoohee for always supporting me.

DEDICATION

To my parents, Sangtae Han and Sunheuy Back, and to my love, Yoohee Choi.

Chapter 1

INTRODUCTION

In recent years, how people use computing devices and the Internet has rapidly evolved. Today, they are connected to the Internet not only when they surf the Web, but many users are almost always online through many different applications over a variety of devices and platforms. These devices include traditional computers, laptops, smartphones, and tablets, as well as newly emerging watches and glasses. Further, the technology trend toward an Internet of Things suggests that more devices around us will have computing power and be connected to the network.

With the increasing number of devices, users have better access to their personal information, such as personal files, location information, contact information, and health information; the types and amount of personal information available through computing devices have ever been increasing. Unfortunately, not only users themselves, but also other parties have better access to user data. When we use applications, remote services store and process user. Many entities are involved in any Internet communication, and not all the entities are trustworthy or reliable. For example, storing personal files in the cloud is convenient, allowing users to access data from multiple devices; however, it also raises potential privacy issues and security risks. It is not surprising to hear a news story that cloud services leak users' personal data [20]. Not only is the reliability of the service an issue, but technology trends and economic forces are moving us toward a world in which personal information is monetized and companies have an incentive to obtain as much of it as possible. Targeted advertising, for example, has revolutionized revenue generation, and tracking user behavior for it is now a common practice for many websites and applications [34, 70].

Despite increasing security and privacy risks and attention from research community

and developers, many issues remain unsolved. This is partly because the diversity of applications and platforms makes it difficult to create a holistic solution. On the other hand, overheads incurred by security and privacy enhancing techniques hurt efficiency of systems (e.g., Tor [23] anonymizing network users suffer from its low performance in network speed[1]), which discourages people from using the solutions.

This dissertation shows that *we can enhance user security and privacy by providing users with control over information exposure within the current untrusted environment.* In the current Internet, users do not have much control over how and which of their information is exposed to other parties and it is irreversible once it occurs. The work presented in this dissertation is to expand user choice by giving them more control to mitigate or eliminate (if possible) the potential security and privacy risk. To do so, we consider the following significant challenges:

- **Untrusted cloud services**: While many applications rely on cloud services to store and process user data, not all services are trustworthy or reliable. In fact, there is no fundamental need for users to trust cloud providers. This dissertation studies this issue with a popular cloud service, file synchronization, and addresses concerns by moving trust to an auditable client and giving users control over *how information is exposed*. For example, the client (configured by the user) can encrypt files before sending, check files' integrity after downloading, and determine where to store them.
- **Linkability of user behavior**: Many users are unaware of that much of their behavior is collected and correlated by remote services. This dissertation discusses this issue in the context of the web browsing; the same principle can be applied to other contexts (e.g., mobile applications). On the web, today's third-party trackers are ubiquitous enough to capture a significant fraction of users' web browsing behavior. While there exist defense mechanisms such as cookie management tools in web browsers, remote services still can correlate users' activity based on unprotected information. Tracking using IP addresses

[1] `https://www.torproject.org/docs/faq.html.en#WhySlow`

is just as accurate as cookies, and with today's protocols, unblockable. To address this issue, we present a cross-layer defense mechanism by introducing an unlinkable *pseudonym* by exploiting IPv6's ample address space, which allows users to have control over *which information can be correlated* at the remote service.

Although we discuss each challenge with specific examples, the principles are more generally applicable to other domains and applications when designing new systems or mitigating security and privacy issues with the existing systems. In the remainder of this chapter, we introduce our approaches addressing the challenges. Next, Chapter 2 and Chapter 3 describe the details of each system. Then, Chapter 4 summarizes the contribution of this dissertation and suggests directions for future research.

1.1 MetaSync: Moving trust to auditable clients

With diverse portable devices, cloud services have become popular as a way to store and process user data. They can provide a shared storage for a user to access the user's files from multiple devices; they can also provide a powerful computing resource to aid limited functionality in resource-constrained devices. One example of such a cloud service is file synchronization. Cloud-based file synchronization services have become tremendously popular. Dropbox reached 400M users in June 2015 and many competing providers offer similar services, including Google Drive, Microsoft OneDrive, Box, and Baidu. These services provide very convenient tools for users, especially given the increasing diversity of user devices needing synchronization. With such resources and tools, mostly available for free, users are likely to upload ever larger amounts of personal and private data.

Unfortunately, not all services are trustworthy or reliable in terms of security and availability. Storage services routinely lose data due to internal faults [11] or bugs [21, 41, 56]. They leak users' personal data [20, 57] and alter user files by adding metadata [12]. They may block access to content (e.g., DMCA takedowns [75]). From time to time, entire cloud services may go out of business (e.g., Ubuntu One [14]).

Our work is based on the premise that users want file synchronization and the storage that existing cloud providers offer, but without the exposure to fragile, unreliable, or insecure services. In fact, there is no fundamental need for users to trust cloud providers, and given the above incidents our position is that users are best served by *not* trusting them. Clearly, a user may encrypt files before storing them in the cloud for confidentiality. More generally, Depot [52] and SUNDR [50] showed how to design systems from scratch in which users of the cloud storage obtain data confidentiality, integrity, and availability without trusting the underlying storage provider. However, these designs rely on fundamental changes to both client and server; our question was whether we could use existing services for these same ends?

In Chapter 2, we introduce MetaSync. Instead of starting from scratch, MetaSync provides file synchronization on top of multiple existing storage providers. We thus leverage resources that are mostly well-provisioned, normally reliable, and inexpensive. While each service provides unique features, their common purpose is to synchronize a set of files between personal devices and the cloud. By combining multiple providers, MetaSync provides users larger storage capacity, but more importantly a more highly available, trustworthy, and higher performance service.

1.2 IPv6 Pseudonym: Flexible control over linkability

Tracking has evolved from individual websites keeping tabs on their users. Today's third-party trackers are capturing a significant fraction of users' web browsing behavior, and they are continuously moving towards wider deployment and even collusion with other services. Google is an extreme example of what is possible, as they theoretically have access to a user's complete logs of email, web history, phone calls, contacts, location information, and much more in addition to being an advertiser. There is no evidence that Google is currently aggregating information on that scale, but the capability exists.

We are not trying to argue that linking of user activities is *always* bad. In fact, it can be a useful tool that benefits both the user and the web service—banks deter fraud

by detecting when a new computer is used to access an account, advertisers support useful services by providing relevant ads/product suggestions, and analytics platforms help websites understand and design for user behavior. However, the line between gathering appropriate information and violating privacy is currently undefined. Our position is that where to draw the line should be up to the user.

While existing defense mechanisms like cookie management systems and other browser tools are able to control tracking to a certain extent, they generally do not handle implicit information like IP addresses. Tracking using IP addresses is just as accurate as cookies, and with today's protocols, unblockable. Lower-level solutions like Tor [23], proxies, or NATs can make IP addresses less meaningful to trackers, but they are coarse-grained, lack flexibility, and in the case of Tor, can hurt performance.

In Chapter 3, we study the linkability issue on the web and present a system to address the concerns. Instead of eliminating the tracking ability of the remote services, our goal is instead to provide users with *more control* over what is linkable and what is not. We do this by co-designing modifications across all relevant software layers in order to protect users' privacy. Central to our design is the concept of a *pseudonym*, which represents a set of user actions and provides an unlinkable abstraction of a single machine.

Chapter 2

METASYNC

This chapter describes a system addressing the first challenge, providing users with more control over the reliability, security and privacy of untrusted cloud services. Specifically, we present the design and implementation of MetaSync, a system to protect users' security and privacy for cloud-based file synchronization services. This work was originally presented in a 2015 paper [36] and also published for broader audience [37].

MetaSync provides file synchronization on top of multiple existing synchronization services. The key challenge is to maintain a globally consistent view of the synchronized files across multiple clients, using only the service providers' unmodified APIs without any centralized server. We assume no direct client-client or server-server communication. To this end, we devise two novel methods: 1) pPaxos, an efficient client-based Paxos algorithm that maintains globally consistent state among multiple passive storage backends, and 2) a stable deterministic replication algorithm that requires minimal reshuffling of replicated objects on service re-configuration, such as increasing capacity or even adding/removing a service.

Putting it all together, MetaSync can serve users better in all aspects as a file synchronization service; users need trust only the software that runs on their own computers. Our prototype implementation of MetaSync, a ready-to-use open source project, currently works with five different file synchronization services, and it can be easily extended to work with other services.

2.1 Goals and Assumptions

The usage model of MetaSync matches that of existing file synchronization services such as Dropbox. A user configures MetaSync with account information for the underlying storage

services, sets up one or more directories to be managed by the system, and shares each directory with zero or more other users. Users can connect these directories with multiple devices (we refer to the devices and software running on them as `clients` in this paper), and local updates are reflected to all connected clients; conflicting updates are flagged for manual resolution. This usage model is supported by a background synchronization daemon (MetaSyncd in Figure 2.1).

For users desiring explicit control over the merge process, we also provide a manual git-like push/pull interface with a command line client. In this case, the user creates a set of updates and runs a script to apply the set. These sets of updates are atomic with respect to concurrent updates by other clients. The system accepts an update only if it has been merged with the latest version pushed by any client.

Our baseline design assumes the backend services to be curious, as well as potentially unreachable, and unreliable. The storage services may try to discover which files are stored along with their content. Some of the services may be unavailable due to network or system failures; some may accidentally corrupt or delete files. However, we assume that service failures are independent, services implement their own APIs correctly (except for losing and corrupting user data), and communications between client and server machines are protected. We also consider extensions to this baseline model where the services have faulty implementations of their APIs or are actively malicious (§2.2.6). Finally, we assume that clients sharing a specific directory are trusted, similar to a shared Dropbox directory today.

With this threat model, the goals of MetaSync are:

- **No direct client-client communication:** Clients coordinate through the synchronization services without any direct communication among clients. In particular, they never need to be online at the same time.
- **Availability:** User files are always available for both read and update despite any predefined number of service outages and even if a provider completely stops allowing any access to its previously stored data.
- **Confidentiality:** Neither user data nor the file hierarchy is revealed to any of the storage

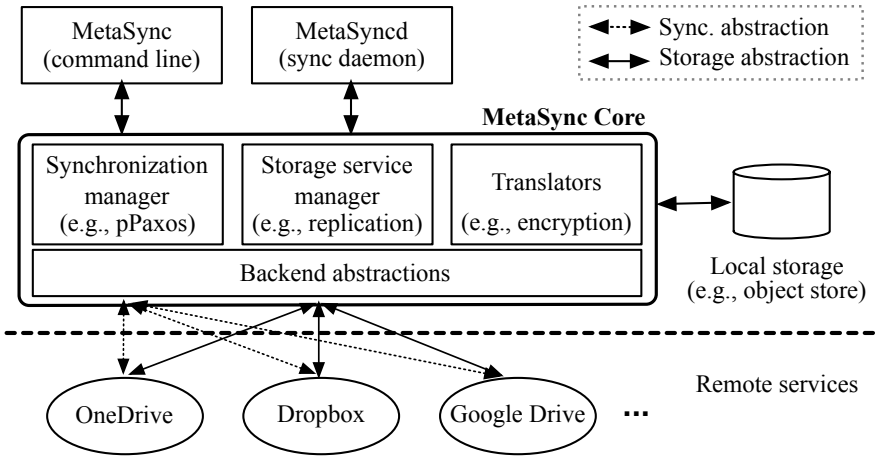

Figure 2.1: **Overview of MetaSync system.** MetaSync has three main components: a storage service manager to coordinate replication; a synchronization manager to orchestrate cloud services; and translators to support data encryption. They are implemented on top of an abstract cloud storage API, which provides a uniform interface to storage backends. MetaSync supports two front-end interfaces: a command line interface and a synchronization daemon for automatic monitoring and check-in.

services. Users may opt out of confidentiality for better performance.

- **Integrity:** The system detects and corrects any corruption of file data by a cloud service, to a configurable level of resilience.
- **Capacity and Performance:** The system should benefit from the combined capacity of the underlying services, while providing faster synchronization and cloning than any individual service.

2.2 System Design

This section describes the design of MetaSync as illustrated by Figure 2.1. MetaSync is a distributed, synchronization system that provides a reliable, globally consistent storage abstraction to multiple clients, by using untrusted cloud storage services. The core library

defines a generic cloud service API; all components are implemented on top of that abstraction. This makes it easy to incorporate a new storage service into our system (§2.2.7). MetaSync consists of three major components: synchronization manager, storage service manager, and translators. The synchronization manager ensures that every client has a consistent view of the user's synchronized files, by orchestrating storage services using pPaxos (§2.2.3). The storage service manager implements a deterministic, stable mapping scheme that enables the replication of file objects with minimal shared information, thus making our system resilient to reconfiguration of storage services (§2.2.4). The translators implement optional modules for encryption and decryption of file objects in services and for integrity checks of retrieved objects, and these modules can be transparently composed to enable flexible extensions (§2.2.5).

2.2.1 Data Management

MetaSync has a similar underlying data structure to that of git [30] in managing files and their versions: objects, units of data storage, are identified by the hash of their content to avoid redundancy. Directories form hash trees, similar to Merkle trees [54], where the root directory's hash is the root of the tree. This root hash uniquely defines a snapshot. MetaSync divides and stores each file into chunks, called Blob objects, in order to maintain and synchronize large files efficiently.

Object store. In MetaSync's object store, there are three kinds of objects—Dir, File and Blob—each uniquely identified by the hash of its content (with an object type as a prefix in Figure 2.2). A File object contains hash values and offsets of Blob objects. A Dir object contains hash values and names of File objects.

In addition to the object store, MetaSync maintains two kinds of metadata to provide a consistent view of the global state: *shared metadata*, which all clients can modify; and *per-client metadata*, which only the single owner (writer) client of the data can modify.

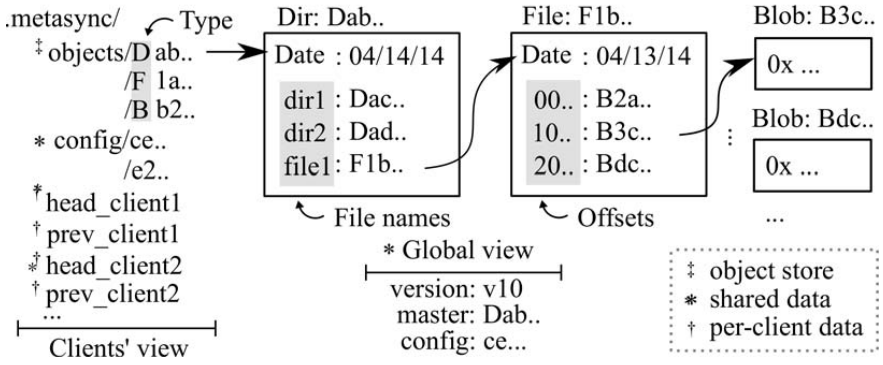

Figure 2.2: **File management in a client's local directory.** The object store maintains user files and directories with content-based addressing, in which the name of each object is based on the hash of its content. MetaSync keeps two kinds of metadata: shared, which all clients update; and per-client, for which the owner client is the only writer. The object store and per-client files can be updated without synchronization, while updates to the shared files require coordinated updates of the backend stores; this is done by the synchronization manager (§2.2.3).

Shared metadata. MetaSync maintains a piece of shared metadata, called `master`, which is the hash value of the root directory in the most advanced snapshot. It represents a consistent view of the global state; every client needs to synchronize its status against the `master`. Another shared piece of metadata is the configuration of the backend services including information regarding the list of backends, their capacities, and authenticators. When updating any of the shared metadata, we invoke a synchronization protocol built from the APIs provided by existing cloud storage providers (§2.2.3).

Per-client data. MetaSync keeps track of clients' states by maintaining a view of each client's status. The per-client metadata includes the last synchronized value, denoted as `prev_clientID`, and the current value representing the client's recent updates, denoted as `head_clientID`. If a client hasn't changed any files since the previous synchronization, the

value of `prev_clientID` is equal to that of `head_clientID`. As this data is updated only by the corresponding client, it does not require any coordinated updates. Each client stores a copy of its per-client metadata into all backends after each update.

2.2.2 Overview

MetaSync's core library maintains the above data structures and exposes a reliable storage abstraction to applications. The role of the library is to mediate accesses and updates to actual files and metadata, and further interacts with the backend storage services to make file data persistent and reliable. The command line wrapper of the APIs works similarly with version control systems.

Initially, a user sets up a directory to be managed by MetaSync; files and directories under that directory will be synchronized. This is equivalent to creating a repository in typical version control systems. Then, MetaSync creates a metadata directory (`.metasync` as shown in Figure 2.2) and starts the synchronization of file data to backend services.

Each managed directory has a name (called namespace) in the system to be used in synchronizing with other clients. Upon initiation, MetaSync creates a folder with the name in each backend. The folder at the backend storage service stores the configuration information plus a subset of objects (§2.2.4). A user can have multiple directories with different configurations and composition of backends.

When files in the system are changed, an update happens as follows: (1) the client updates the local objects and `head_client` to point to the current root (§2.2.1); (2) stores the updated data blocks on the appropriate backend services (§2.2.4)); and (3) proposes its `head_client` value as the new value for `master` using pPaxos (§2.2.3)). The steps (1) and (2) do not require any coordination, as (1) happens locally and (2) proceeds asynchronously. Note that these steps are provided as separate functions to applications, thus each application or user can decide when to run each step; crucially, a client does not have to update global `master` for every local file write.

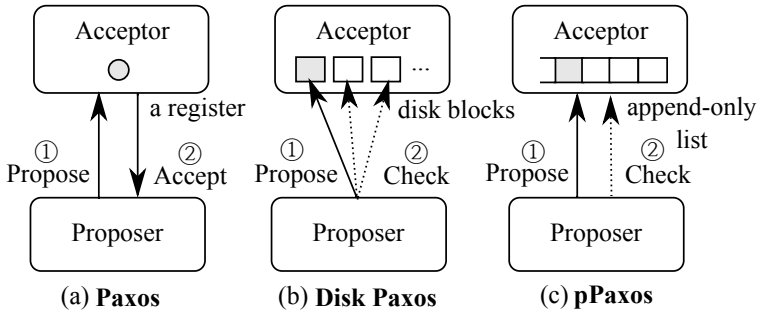

Figure 2.3: **Comparison of operations between a proposer and an acceptor in Paxos, Disk Paxos, and pPaxos.** Each acceptor in Paxos makes a local decision to accept or reject a proposal and then replies with the result. Disk Paxos assumes acceptors are passive; clients write proposals into per-client disk blocks at each acceptor. Proposers need to check every per-client block (at every acceptor) to determine if their proposal was accepted, or preempted by another concurrent proposal. With pPaxos, the append-only log allows clients to efficiently check the outcome at the passive acceptor.

2.2.3 Consistent Update of Global View: pPaxos

The file structure described above allows MetaSync to minimize the use of synchronization operations. Each object in the object store can be independently uploaded as it uses content-based addressing. Each per-client data (e.g., `head_client_*`) is also independent since we ensure that only the owning client modifies the file. Thus, synchronization to avoid potential race conditions is necessary only when a client wants to modify shared data (i.e., `master` and configuration).

pPaxos. In a distributed environment, it is not straightforward to coordinate updates to data that can be modified by multiple clients simultaneously. To create a consistent view, clients must agree on the sequence of updates applied to the shared state, by agreeing on the next update applied at a given point.

Clients do not have communication channels between each other (e.g., they may be

offline), so they need to rely on the storage services to achieve this consensus. However, these services do not communicate with each other, nor do they implement consensus primitives. Instead, we devise a variant of Paxos [49], called pPaxos (passive Paxos) that uses the exposed APIs of these services.

We start our overview of pPaxos by relating it to the classic Paxos algorithm (see Figure 2.3(a)). There, each client works as a proposer and learner; the next state is determined when a majority accepts a given proposal. Acceptors act in concert to prevent inconsistent proposals from being accepted; failing proposals are retried. We cannot assume that the backend services will implement the Paxos acceptor algorithm. Instead, we only require them to provide an *append-only list* that *atomically* appends an incoming message at the end of the list. This abstraction is either readily available or can be layered on top of the interface provided by existing storage service providers (Table 2.3). With this append-only list abstraction, backend services can act as *passive acceptors*. Clients determine which proposal was "accepted" by examining the log of messages to determine what a normal Paxos acceptor would have done.

Algorithm. With an append-only list, pPaxos becomes a simple adaptation of classic Paxos, where the decision as to what proposal was accepted is performed by proposers. Each client keeps a data structure for each backend service, containing the state it would have if it processed its log as a Paxos acceptor (Figure 2.4 Lines 1-4). To propose a value, a client sends a `PREPARE` to every storage backend with a proposal number (Lines 7-8); this message is appended to the log at every backend. The proposal number must be unique (e.g., client IDs are used to break ties). The client determines the result of the prepare message by fetching and processing the logs at each backend (Lines 25-29). It aborts its proposal if another client inserted a larger proposal number in the log (Line 10). As in Paxos, the client proposes as the new root the value in the highest numbered proposal "accepted" by any backend server (Lines 12-15), or its own new root if none has been accepted. It sends this value in an `ACCEPT_REQ` message to every backend (Lines 18-19) to be appended to its

```
1:  struct Acceptor
2:      round:      promised round number
3:      accepted:   all accepted proposals
4:      backend:    associated backend service
    [Proposer]
5:  procedure ProposeRound(value, round, acceptors)
      prepare:
6:    concurrently
7:    for all a ← acceptors do
8:      Send(⟨PREPARE, round⟩ → a.backend)
9:      Update(a)
10:     if a.round > round then abort
11:   wait until done by a majority of acceptors
      accept:
12:   accepted ← ∪_{a∈acceptors} a.accepted
13:   if |accepted| > 0 then
14:     p ← arg max{p.round | p ∈ accepted}
15:     value ← p.value
16:   proposal ← ⟨round, value⟩
17:   concurrently
18:   for all a ← acceptors do
19:     Send(⟨ACCEPT_REQ, proposal⟩ → a.backend)
20:     Update(a)
21:     if proposal ∉ a.accepted then abort
22:   wait until done by a majority of acceptors
      commit:
23:   return proposal
24: procedure Update(acceptor)
25:   log ← FetchNewLog(acceptor.backend)
26:   for all msg ∈ log do
27:     switch msg do
28:       case ⟨PREPARE, round⟩
29:         acceptor.round ← max(round, acceptor.round)
30:       case ⟨ACCEPT_REQ, proposal⟩
31:         if proposal.round ≥ acceptor.round then
32:           acceptor.accepted.append(proposal)
33: procedure OnRestartAfterFailure(round)
34:   IncreaseRound
35:   WaitExponentially
36:   ProposeRound(value, round, acceptors)
    [Passive Acceptor]
37: procedure OnNewMessage(⟨msg, round⟩)
38:   Append(⟨msg, round⟩ → log)
```

Figure 2.4: **pPaxos Algorithm**

log; the value is committed if no higher numbered `PREPARE` message intervenes in the log (Lines 20-21, 30-32). When the new root is accepted by a majority, the client can conclude it has committed the new updated value (Line 23). In case it fails, to avoid repeated conflicts the client chooses a random exponential back-off and tries again with an increased proposal number (Lines 33-36).

This setting is similar to the motivation behind Disk Paxos [29]; indeed, pPaxos can be considered as an optimized version of Disk Paxos (Figure 2.3(b)). Disk Paxos assumes that the storage device provides only a simple block interface. Clients write proposals to their own block on each server, but they must check everyone else's blocks to determine the outcome. Thus, Disk Paxos takes time proportional to the product of the number of servers

APIs	Description
propose(`prev`, `next`)	Propose a next value of `prev`. It returns the accepted next value, which could be `next` or some other value proposed by another client.
get_recent()	Retrieve the most recent value.

Table 2.1: **Abstractions for consistent update**

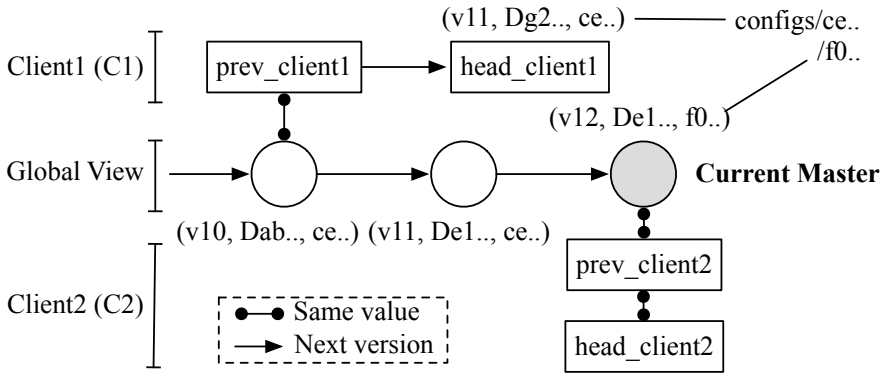

Figure 2.5: **An example snapshot of pPaxos status with two clients.** Each circle indicates a pPaxos instance. C1 synchronized against v10. It modified some files but the changes have not been synchronized yet `head_client1`. C2 changed some files and the changes were made into v11, then made changes in configuration and synchronized it (v12). Then, it hasn't made any changes. If C1 tries to propose the next value of v10 later, it fails. It needs to merge with v12 and creates v13 head. In addition, C1 can learn configuration changes when getting v12.

and clients; pPaxos is proportional to number of servers.

pPaxos in action. MetaSync maintains two types of shared metadata: the `master` hash value and service configuration. Unlike a regular file, the configuration is replicated in all backends (in their object stores). Then, MetaSync can uniquely identify the shared data

with a three tuple: `(version, master_hash, config_hash)`.

Version is a monotonically increasing number which is uniquely determined for each `master_hash`, `config_hash` pair. This tuple is used in pPaxos to describe the status of a client and is stored in `head_client` and `prev_client`.

The pPaxos algorithm explained above can determine and store the next value of the three tuple. Then, we build the functions listed in Table 2.1 by using a pPaxos instance per synchronized value. Each client keeps the last value with which it synchronized (`prev_client`). To proposes a new value, the client runs pPaxos to update the previous value with the new value. If another value has already been accepted, it can try to update the new value after merging with it. It can repeat this until it successfully updates the master value with its proposed one. This data structure can be logically viewed as a linked list, where each entry points the next hash value, and the tail of the list is the most up-to-date. Figure 2.5 illustrates an example snapshot of pPaxos status.

Merging. Merging is required when a client synchronizes its local changes (`head`) with the current master that is different from what the client previously synchronized (`prev`). In this case, proposing the current head as the next update to `prev` returns a different value than the proposed head as other clients have already advanced the `master` value. The client has to merge its changes with the current `master` into its head. To do this, MetaSync employs three-way merging as in other version control systems. This allows many conflicts to be automatically resolved. Of course, three-way merging cannot resolve all conflicts, as two clients may change the same parts of a file. In our current implementation, for example, MetaSync generates a new version of the file with `.conflict.N` extension, which allows for users to resolve the conflict manually.

2.2.4 Replication: Stable Deterministic Mapping

MetaSync replicates objects (in the object store) redundantly across R storage providers (R is configurable, typically $R = 2$) to provide high availability even when a service is temporarily

inaccessible. This also provides potentially better performance over wide area networks. Since R is less than the number of services, it is required to maintain information regarding the mapping of objects to services. In our settings, where the storage services passively participate in the coordination protocol, it is particularly expensive to provide a consistent view of this shared information. Not only that, MetaSync requires a mapping scheme that takes into account storage space limits imposed by each storage service; if handled poorly, lack of storage at a single service can block the entire operation of MetaSync, and typical storage services vary in the (free) space they provide, ranging from 2 GB in Dropbox to 2 TB in Baidu. In addition, the mapping scheme should consider a potential reconfiguration of storage services (e.g., increasing storage capacity or adding/removing a backend service); upon changes, the re-balancing of distributed objects should be minimal.

Goals. Instead of maintaining the mapping information of each object, we use a stable, deterministic mapping function that locates each object to a group of services over which it is replicated; each client can calculate the same result independently given the same object. Given a hash of an object (mod H), the mapping is: map: $H \rightarrow \{s : |s| = R, s \subset S\}$, where H is the hash space, S is the set of services, and R is the number of replicas. This mapping should meet three requirements:

R1 Support variations in storage size limits across different services and across different users.

R2 Share minimal information amongst services.

R3 Minimize realignment of objects upon removal or addition of a service.

To provide a balanced mapping that takes into account of storage variations of each service (R1), we may use a mapping scheme that represents storage capacity as the number of virtual nodes in a consistent hashing algorithm [43, 72]. Since it deterministically locates each object onto an identifier circle in the consistent hashing scheme, MetaSync can minimize information shared among storage providers (R2).

1: **procedure** INIT($Services$, H) ▷ H: HashSpace size, bigger values produce better mappings
2: $N \leftarrow \{(sId, vId) : sId \in Services, 0 \leq vId < \text{Cap}(sId)\}$ ▷ Cap: normalized capacity of the service
3: **for all** $i < H$ **do** map[i] = Sorted(N, $key = md5(i, sId, vId)$)
4: **return** map
5: **procedure** GETMAPPING($object, R$)
6: $i \leftarrow hash(object) \bmod H$
7: **return** Uniq(map[i], R) ▷ Uniq: the first R distinct services

Figure 2.6: **The deterministic mapping algorithm.**

However, using consistent hashing in this way has two problems: an object can be mapped into a single service over multiple vnodes, which reduces availability even though the object is replicated, and a change in service's capacity—changing the number of virtual nodes, so the size of hash space—requires to reshuffle all the objects distributed across service providers (R3). To solve these problems, we introduce a new, stable deterministic mapping scheme that maps an object to a unique set of virtual nodes and also minimizes reshuffling upon any changes to virtual nodes (e.g., changes in configurations). This construction is challenging because our scheme should randomly map each service to a virtual node and balance object distribution, but at the same time, be stable enough to minimize remapping of replicated objects upon any change to the hashing space. The key idea is to achieve the random distribution via hashing, and achieve stability of remapping by sorting these hashed values; for example, an increase of storage capacity will change the order of existing hashed values by at most one.

Algorithm. Our stable deterministic mapping scheme is formally described in Figure 2.6. For each backend storage provider, it utilizes multiple virtual storage nodes, where the number of virtual nodes per provider is proportional to the storage capacity limit imposed by the provider for a given user. The concept of virtual nodes is similar to that used in systems such as Dynamo [22]. Then it divides the hash space into H partitions. H

is configurable, but remains fixed even as the service configuration changes. H can be arbitrary but need to be larger than the sum of normalized capacity, with larger values producing better-balanced mappings for heterogeneous storage limits. During initialization, the mapping scheme associates differently ordered lists of virtual nodes with each of the H partitions. The ordering of the virtual nodes in the list associated with a partition is determined by hashing the index of the partition, the service ID, and the virtual node ID. Given an object hash n, the mapping returns the first R *distinct* services from the list associated with the (n mod H)th partition.

The mapping function takes as input the set of storage providers, the capacity settings, value of H, and a hash function. Thus, it is necessary to share only these small pieces of information in order to reconstruct this mapping across different clients sharing a set of files. The list of services and the capacity limits are part of the service configuration and shared through the `config` file. The virtual node list is populated proportionally to service capacity, and the ordering in each list is determined by a uniform hash function. Thus, the resulting mapping of objects onto services should be proportional to service capacity limits with large H. Lastly, when N nodes are removed from or added to the service list, an object needs to be newly replicated into at most N nodes.

Our mapping algorithm can be considered as an extension to Highest Random Weight Hashing (HRW) or Rendezvous Hashing [73] where the weight in HRW is defined as $-1 \times hash(object, sID, vID)$ from our hash algorithm and we add virtual nodes and extend it to choosing multiple service nodes for a single object. Such hash algorithm could be useful to other applications as well. Recently, Maglev [26] proposes a new mapping algorithm based on their Maglev consistent hashing in order to achieve better load balancing while sacrificing a bit of minimal disruption (minimal realignment). Their algorithm shares much idea with ours whereas Marglev creates a permutation of numbers in the hash space per node and our mapping algorithm creates a permutation of nodes per hash value.

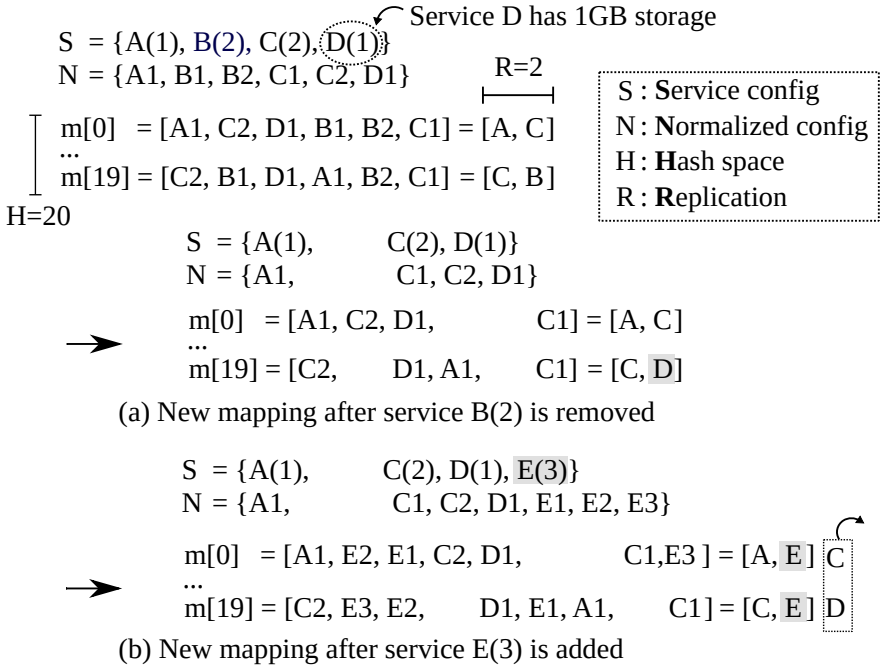

Figure 2.7: **An example of deterministic mapping and its reconfigurations.** The initial mapping is deterministically generated by Figure 2.6, given the configuration of four services, $A(1), B(2), C(2), D(1)$ where the number represents the capacity of each service. (a) and (b) show new mappings after configuration is changed. The grayed mappings indicate the new replication upon reconfiguration, and the dotted rectangle in (b) represents replications that will be garbage collected.

Example. Figure 2.7 shows an example of our mapping scheme with four services ($|S| = 4$) providing 1GB or 2GB of free spaces–for example, A(1) means that service A provides 1GB of free space. Given the replication requirement ($R = 2$) and the hash space ($H = 20$), we can populate the initial mapping with `Init` function from Figure 2.6. Subfigures (a) and (b) illustrate the realignment of objects upon the removal of service B(2) and the inclusion of a new service E(3).

2.2.5 Translators

MetaSync provides a plugin system, called Translators, for encryption and integrity check. Translators is highly modular so can easily be extended to support a variety of other transformations such as compression. Plugins should implement two interfaces, `put` and `get`, which are invoked before storing objects to and after retrieving them from backend services. Plugins are chained, so that when an object is stored, MetaSync invokes a chain of `put` calls in sequence. Similarly, when an object is retrieved, it goes through the same chain but in reverse.

Encryption translator is currently implemented using a symmetric key encryption (AES-CBC). MetaSync keeps the encryption key locally, but does not store on the backends. When a user clones the directory in another device, the user needs to provide the encryption key. Integrity checker runs hash function over retrieved object and compares the digest against the file name. If it does not match, it drops the object and downloads the object by using other backends from the mapping. It needs to run only in the `get` chain.

2.2.6 Fault Tolerance

To operate on top of multiple storage services that are often unreliable (they are free!), faulty (they scan and tamper with your files), and insecure (some are outside of your country), MetaSync should tolerate faults.

Data model. By replicating each object into multiple backends (R in §2.2.4), MetaSync can tolerate loss of file or directory objects, and tolerate temporal unavailability or failures of $R - 1$ concurrent services.

File integrity. Similarly with other version control systems [30], the hash tree ensures each object's hash value is valid from the root (`master, head`). Then, each object's integrity can be verified by calculating the hash of the content and comparing with the name when it is retrieved from the backend service. The value of `master` can be signed to protect against

tampering. When MetaSync finds an altered object file, it can retrieve the data from another replicated service through the deterministic mapping.

Consistency control. MetaSync runs pPaxos for serializing updates to the shared value for `config` and `master`. The underlying pPaxos protocol requires $2f + 1$ acceptors to ensure correctness if f acceptors may fail under the fail-stop model.

Byzantine Fault Tolerant pPaxos. pPaxos can be easily extended to make it resilient to other forms of service failures, e.g., faulty implementations of the storage service APIs and even actively malicious storage services. Note that even with Byzantine failures, each object is protected in the same way through replication and integrity checks. However, updates of global view need to be handled more carefully. We assume that clients are trusted and work correctly, but backend services may have Byzantine behavior. When sending messages for proposing values, a client needs to sign it. This ensures that malicious backends cannot create arbitrary log entries. Instead, the only possible malicious behavior is to break consistency by omitting log entries and reordering them when clients fetch them; a backend server may send any subset of the log entries in any order. Under this setting, pPaxos works similarly with the original algorithm, but it needs $3f+1$ acceptors when f may concurrently fail. Then, for each prepare or accept, a proposing client needs to wait until $2f+1$ acceptors have prepared or accepted, instead of $f + 1$. It is easy to verify the correctness of this scheme. When a proposal gets $2f+1$ accepted replies, even if f of the acceptors are Byzantine, the remaining $f+1$ acceptors will not accept a competing proposal. As a consequence, competing proposals will receive at most $2f$ acceptances and will fail to commit. Note that each file object is still replicated at only $f + 1$ replicas, as data corruption can be detected and corrected as long as there is a single non-Byzantine service. As a consequence, the only additional overhead of making the system tolerate Byzantine failures is to require a larger quorum $(2f + 1)$ and a larger number of storage services $(2f + 1)$ for implementing the synchronization operation associated with updating `master`.

APIs	Description
(a) Storage abstraction	
get(`path`)	Retrieve a file at `path`
put(`path`, `data`)	Store `data` at `path`
delete(`path`)	Delete a file at `path`
list(`path`)	List all files under `path` directory
poll(`path`)	Check if `path` was changed
(b) Synchronization abstraction	
append(`path`, `msg`)	Append `msg` to the list at `path`
fetch(`path`)	Fetch a log from `path`

Table 2.2: **Abstractions for backend storage services.**

2.2.7 Backend abstractions

Storage abstraction. Any storage service having an interface to allow clients to read and write files can be used as a storage backend of MetaSync. More specifically, it needs to provide the basis for the the functions listed in Table 2.2(a). Many storage services provide a developer toolkit to build a customized client accessing user files [24, 32]; we use these APIs to build MetaSync. Not only cloud services provide these APIs, it is also straightforward to build these functions on user's private servers through SSH or FTP. MetaSync currently supports backends with the following services: Dropbox, GoogleDrive, OneDrive, Box.net, Baidu, and local disk.

Synchronization abstraction. To build the primitive for synchronization, an append-only log, MetaSync can use any services that provide functions listed in Table 2.2(b). How to utilize the underlying APIs to build the append-only log varies across services. We summarize how MetaSync builds it for each provider in Table 2.3.

2.2.8 Other Issues

Sharing. MetaSync allows users to share a folder and work on the folder. While not many backend services have APIs for sharing functions—only Google Drive and Box have it among services that we used—others can be implemented through browser emulation. Once sharing invitation is sent and accepted, synchronization works the same way as in the one-user case. If files are encrypted, we assume that all collaborators share the encryption key.

Collapsing directory. All storage services manage individual files for uploading and downloading. As we see later in Table 2.4, throughput for uploading and downloading small files are much lower than those for larger files. As an optimization, we collapse all files in a directory into a single object when the total size is small enough.

2.3 Implementation

We have implemented a prototype of MetaSync in Python, and the total lines of code is about 7.5K. The current prototype supports five backend services including Box, Baidu, Dropbox, Google Drive and OneDrive, and works on all major OSes including Linux, Mac and Windows. MetaSync provides two front-end interfaces for users, a command line interface similar to git and a synchronization daemon similar to Dropbox.

Abstractions. Storage services provide APIs equivalent to MetaSync's `get()` and `put()` operations defined in Table 2.2. Since each service varies in its support for the other operations, we summarize the implementation details of each service provider in Table 2.3. For implementing synchronization abstractions, `append()` and `fetch()`, we utilized the *commenting* features in Box, Google and OneDrive, and *versioning* features in Dropbox. If a service does not provide any efficient ways to support synchronization APIs, MetaSync falls back to the default implementation of those APIs that are built on top of their storage APIs, described for Baidu in Table 2.3. Note that for some services, there are multiple ways to implement the synchronization abstractions. In that case, we chose to use mechanisms with

Service	Synchronization API		Storage API
	`append(path, msg)`	`fetch(path)`	`poll(path)`
Box Google OneDrive	Create an empty `path` file and add `msg` as *comments* to the `path` file.	Download the entire comments attached on the `path` file.	Use `events` API, allowing long polling. (Google, OneDrive: periodically list `pPaxos` directory to see if any changes)
Baidu	Create a `path` directory, and consider each file as a log entry containing `msg`. For each entry, we create a file with an increasing sequence number as its name. If the number is already taken, we will get an exception and try with a next number.	List the `path` directory, and download new log entries since last fetch (all files with subsequent sequence numbers).	Use `diff` API to monitor if there is any change over the user's drive.
Dropbox	Create a `path` file, and overwrite the file with a new log entry containing `msg`, relying on Dropbox's versioning.	Request a list of versions of the `path` file.	Use `longpoll_delta`, a blocked call, that returns if there is a change under `path`.
Disk†	Create a `path` file, and append `msg` at the end of the file.	Read the new entries from the `path` file.	Emulate long polling with a condition variable.

Table 2.3: **Implementation details of synchronization and storage APIs for each service.** Note that implementations of other storage APIs (e.g., `put()`) can be directly built with APIs provided by services, with minor changes (e.g., supporting namespace). Disk† is implemented for testing.

better performance.

Front-ends. The MetaSync daemon monitors file changes by using `inotify` in Linux, `FSEvents` and `kQueue` in Mac and `ReadDirectoryChangesW` in Windows, all abstracted by the Python library `watchdog`. Upon notification, it automatically uploads detected changes into backend services. It batches consecutive changes by waiting 3 more seconds after notification so that all modified files are checked in as a single commit to reduce synchronization overhead. It also polls to find changes uploaded from other clients; if so, it merges them into the local drive. The command line interface allows users to manually manage and synchro-

nize files. The usage of MetaSync commands is similar to that of version control systems (e.g., `metasync init`, `clone`, `checkin`, `push`).

2.4 Evaluation

This section answers the following questions:

- What are the performance characteristics of pPaxos?
- How quickly does MetaSync reconfigure mappings as services are added or removed?
- What is the end-to-end performance of MetaSync?

Each evaluation is done on Linux servers connected to campus network except for synchronization performance in §2.4.3. Since most services do not have native clients for Linux, we compared synchronization time for native clients and MetaSync on Windows desktops.

Before evaluating MetaSync, we measured the performance variance of services in Table 2.4 via their APIs. One important observation is that all services are slow in handling small files. This provides MetaSync the opportunity to outperform them by combining small objects.

2.4.1 pPaxos performance

We measure how quickly pPaxos reaches consensus as we vary the number of concurrent proposers. The results of the experiment with 1-5 proposers over 5 storage providers are shown in Figure 2.8. A single run of pPaxos took about 3.2 sec on average under a single writer model to verify acceptance of the proposal when using all 5 storage providers. This requires at least four round trips: `PREPARE` (Send, FetchNewLog) and `ACCEPT_REQ` (Send, FetchNewLog) (Figure 2.4) (there could be multiple rounds in FetchNewLog depending on the implementation for each service). It took about 7.4 sec with 5 competing proposers. One important thing to emphasize is that, even with a slow connection to Baidu, pPaxos can quickly be completed with a single winner of that round. Also note that when compared to a single storage provider, the latency doesn't degrade with the increasing number of storage

Services	1 KB		1 MB		10 MB		100 MB	
	U.S.	China	U.S.	China	U.S.	China	U.S.	China
Baidu	0.7 / 0.8	1.8 / 2.6	0.21 / 0.22	0.12 / 1.48	0.22 / 0.94	0.13 / 2.64	0.24 / 1.07	0.13 / 3.38
Box	1.4 / 0.6	0.8 / 0.2	0.73 / 0.44	0.11 / 0.12	4.79 / 3.38	0.13 / 0.68	17.37 / 15.77	0.13 / 1.08
Dropbox	1.2 / 1.3	0.5 / 0.5	0.59 / 0.69	0.10 / 0.20	2.50 / 3.48	0.09 / 0.41	3.86 / 14.81	0.13 / 0.68
Google	1.4 / 0.8	-	1.00 / 0.77	-	5.80 / 5.50	-	9.43 / 26.90	-
OneDrive	0.8 / 0.5	0.3 / 0.1	0.45 / 0.34	0.01 / 0.05	3.13 / 2.08	0.11 / 0.12	7.89 / 6.33	0.11 / 0.44
	KB/s		MB/s		MB/s		MB/s	

Table 2.4: **Upload and download bandwidths of four different file sizes on each service from U.S. and China.** This preliminary experiment explains three design constrains of MetaSync. First, all services are extremely slow in handling small files, 7k/34k times slower in uploading/downloading 1 KB files than 100 MB on Google storage service. Second, the bandwidth of each service approaches its limit at 100 MB. Third, performance varies with locations, 30/22 times faster in uploading/downloading 100 MB when using Dropbox in U.S. compared to China.

providers—it is slower than using a certain backend service (Google), but it is similar to the median case as the latency depends on the proposer getting responses from the majority.

Next, we compare the latency of a single round for pPaxos with that for Disk Paxos [29]. We build Disk Paxos with APIs by assigning a file as a block for each client. Figure 2.9 shows the results with varying number of clients when only one client proposes a value. As we explain in §2.2.3, Disk Paxos gets linearly slower with increasing number of clients even when all other clients are inactive, since it must read the current state of all clients.

2.4.2 Deterministic mapping

We then evaluate how fairly our deterministic mapping distributes objects into storage services with different capacity, in three replication settings ($R = 1, 2$). We test our scheme by synchronizing source tree of Linux kernel 3.10.38, consisting of a large number of small files (464 MB), to five storage services, as detailed in Table 2.5. We use H = (5×sum of

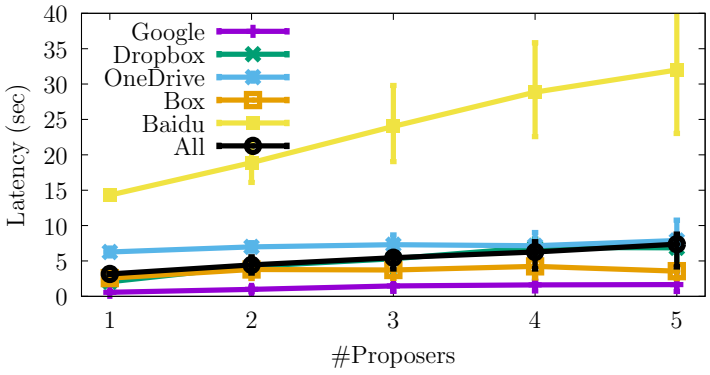

Figure 2.8: **Latency to run a single pPaxos round.** When using 5 different storage providers as backend nodes (all), the common path of pPaxos at a single proposer takes 3.2 sec, and the slow path with 5 competing proposers takes 7.4 sec in median. Each measurement is done 5 times, and it shows the average latency.

normalized space) $= 10,410$ for this testing. In $R = 1$, where we upload each object once, MetaSync locates objects in balance to all services—it uses 0.02% of each service's capacity consistently. However, since Baidu provides 2TB (98% of MetaSync's capacity in this configuration), most of the objects will be allocated into Baidu. This situation improves for $R = 2$, since objects will be placed into other services beyond Baidu. Baidu gets only 6.2 MB of more storage when increasing $R = 1 \rightarrow 2$, and our mapping scheme preserves the balance for the rest of services (using 1.3%).

The entire mapping plan is deterministically derived from the shared `config`. The size of information to be shared is small (less than 50B for the above example), and the size of the populated mapping is about 3MB.

The relocation scheme is resilient to changes as well, meaning that redistribution of objects is minimal. As in Table 2.6, when we increased the configured replication by one ($R = 2 \rightarrow 3$) with 4 services, MetaSync replicated 193 MB of objects in about half a minute. When we removed a service from the configuration, MetaSync redistributed 96.5 MB of objects in about 20 sec. After adding and removing a storage backend, MetaSync needs

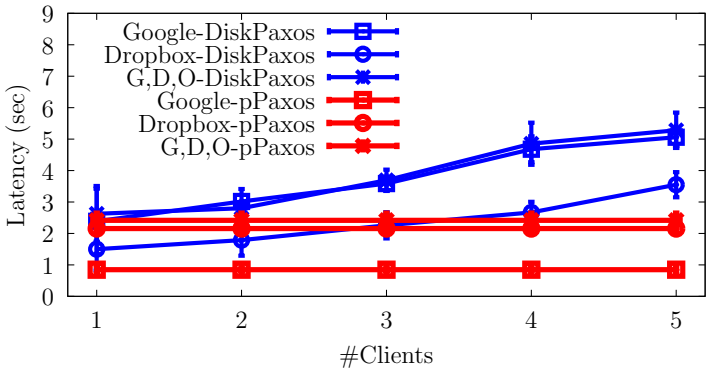

Figure 2.9: **Comparison of latency to run a single round for Disk Paxos and pPaxos.** It shows the latency with varying number of clients when only one client proposes a value. Each line represents different backend setting; G,D,O: Google, Dropbox, and Onedrive. While pPaxos is not affected by the number of clients, Disk Paxos latency increases with it. Each measurement is done 5 times, and it shows the average latency.

to delete redundant objects from the previous configuration, which took 40.6/14.7 sec for removing/adding OneDrive in our experiment. However, the garbage collection will be asynchronously initiated during idle time.

2.4.3 End-to-end performance

We selected three workloads to demonstrate performance characteristics. First, Linux kernel source tree (2.6.1) represents the most challenging workload for all storage services due to its large volume of files and directory (920 directories and 15k files, total 166 MB). Second, MetaSync's paper represents a causal use of synchronization service for users (3 directories and 70 files, total 1.6 MB). Third, sharing photos is for maximizing the throughput of each storage service with bigger files (50 files, total 193 MB).

Table 2.7 summarizes our results for end-to-end performance for all workloads, comparing MetaSync with the native clients provided by each service. Each workload was copied into one client's directory before synchronization is started. The synchronization time was measured

Repl.	Dropbox (2 GB)	Google (15 GB)	Box (10 GB)	OneDrive (7 GB)	Baidu (2048 GB)
$R = 1$	77 (0.09%)	660 (0.75%)	475 (0.54%)	179 (0.20%)	86,739 (98.42%)
	0.34 MB (0.02%)	2.87 MB (0.02%)	2.53 MB (0.02%)	0.61 MB (0.01%)	463.8 MB (0.02%)
$R = 2$	5,297 (3.01%)	39,159 (22.22%)	25,332 (14.37%)	18,371 (10.42%)	88,101 (49.98%)
	27.4 MB (1.34%)	206.4 MB (1.34%)	138.2 MB (1.35%)	98.3 MB (1.37%)	470.0 MB (0.02%)
$R = 3$	13,039 (4.93%)	66,964 (25.33%)	54,505 (20.62%)	41,752 (15.79%)	88,130 (33.33%)
	67.2 MB (3.28%)	355.7 MB (2.32%)	294.8 MB (2.88%)	222.7 MB (3.11%)	470.1 MB (0.02%)

Table 2.5: **Replication results by our deterministic mapping scheme.** We synchronized files of Linux kernel 3.10.38 (Table 2.7) on 5 different services with various storage space, given for free. It comprises total 470 MB of files, consisting of 88k objects, and files are replicated across all storage backends. Note that for this mapping test, we turned off the optimization of collapsing directories. Our deterministic mapping distributed objects in balance: for example, in $R = 2$, Dropbox, Google, Box and OneDrive used consistently 1.35% of their space, even with 2-15 GB of capacity variation. Also, $R = 1$ approaches to the perfect balance, using 0.02% of storage space in all services.

as the length of interval between when one desktop starts to upload files and the creation time of the last file synced on the other desktop. We also measured the synchronization time for all workloads by using MetaSync with different settings. MetaSync outperforms any individual service for all workloads. Especially for Linux kernel source, it took only 12 minutes when using 4 services (excluding Baidu located outside of the country) compared to more than 2 hrs with native clients. This improvement is possible due to using concurrent connections to multiple backends, and optimizations like collapsing directories. Although these native clients may not be optimized for the highest possible throughput, considering that they may run as a background service, it would be beneficial for users to have a faster option. It is also worth noting that replication helps sync time, especially when there is a slower service, as shown in the case with $S = 5, R = 1, 2$; a downloading client can use faster

Reconfiguration	#Objects Added / Removed	Time (sec) Replication / GC
$S = 4, R = 2 \rightarrow 3$	101 / 0	33.7 / 0.0
$S = 4 \rightarrow 3, R = 2$	54 / 54	19.6 / 40.6
$S = 3 \rightarrow 4, R = 2$	54 / 54	29.8 / 14.7

Table 2.6: **Time to relocate 193 MB amount of objects.** We relocated the photo-sharing workloads in Table 2.7 on increasing the replication ratio, removing an existing service, and adding one more service. MetaSync quickly re-balances its mapping (and replication) based on its new `config`. We used four services, Dropbox, Box, GoogleDrive, and OneDrive ($S = 4$) for experimenting with the replication, including ($S = 3 \rightarrow 4$) and excluding OneDrive ($S = 4 \rightarrow 3$) for re-configuring storage services.

services while an uploading client can upload a copy in the background.

Clone. Storage services often limit their download throughput: for example, MetaSync can download at 5.1 MB/s with Dropbox as a backend, and at 3.4 MB/s with Google Drive, shown in Figure 2.10. Note that downloading is done already by using concurrent connections even to the same service. By using multiple storage services, MetaSync can fully exploit the bandwidth of local connection of users, not limited by the allowed throughput of each service. For example, MetaSync with both services and R=2 took 25.5 sec for downloading 193 MB data, which is at 7.6 MB/s.

2.5 Related Work

A major line of related work, starting with Farsite [6] and SUNDR [50] but carrying through SPORC [28], Frientegrity [27], and Depot [52], is how to provide tamper resistance and privacy on untrusted storage server nodes. These systems assume the ability to specify the client-server protocol, and therefore cannot run on unmodified cloud storage services. A

Workload	**Dropbox**	**Google**	**Box**	**OneDrive**	**Baidu**	**MetaSync**			
# Services						5	5	4	4
# Repl.						1	2	1	2
Linux kernel source	2h 45m	> 3hrs	> 3hrs	2h 03m	> 3hrs	1h 8m	13m 51s	18m 57s	12m 18s
MetaSync paper	48	42	148	54	143	55	50	27	26
Photo sharing	415	143	536	1131	1837	1185	180	137	112

Table 2.7: **Synchronization performance of 5 native clients provided by each storage service, and with four different settings of MetaSync.** For $S = 5, R = 1$, using all of 5 services without replication, MetaSync provides comparable performance to native clients–median speed for MetaSync paper and photo sharing, but outperforming for Linux kernel workloads. However, for $S = 5, R = 2$ where replicating objects twice, MetaSync outperform >10 times faster than Dropbox in Linux kernel and 2.3 times faster in photo sharing; we can finish the synchronization right after uploading a single replication set (but complete copy) and the rest will be scheduled in background. To understand how slow straggler (Baidu) affects the performance ($R = 1$), we also measured synchronization time on $S = 4$ without Baidu, where MetaSync vastly outperforms all services.

further issue is equivocation; servers may tell some users that updates have been made, and not others. Several of these systems detect and resolve equivocations after the fact, resulting in a weaker consistency model than MetaSync's linearizable updates. A MetaSync user knows that when a push completes, that set of updates is visible to all other users and no conflicting updates will be later accepted. Like Farsite, we rely on a stronger assumption about storage system behavior—that failures across multiple storage providers are independent, and this allows us to provide a simpler and more familiar model to applications and users.

Likewise, several systems have explored composing a storage layer on top of existing storage systems. Syndicate [61] is designed as an API for applications; thus, they delegate design choices such as how to manage files and replicate to application policy. SCFS [10] implements a sharable cloud-backed file system with multiple cloud storage services. Unlike

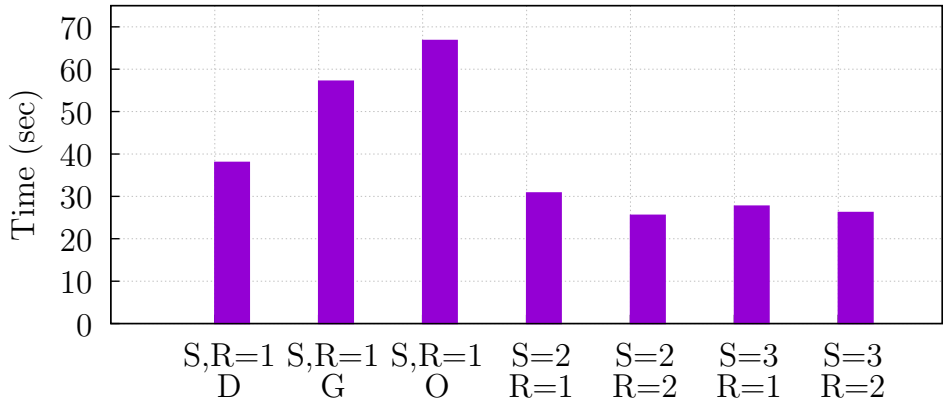

Figure 2.10: **Time to clone 193 MB photos.** When using individual services as a back-end (Dropbox, Google, and OneDrive), MetaSync took 40-70 sec to clone, but improved the performance, 25-30 sec (30%) by leveraging the distributions of objects across multiple services.

MetaSync, Syndicate and SCFS assume separate services for maintaining metadata and consistency. RACS [5] uses RAID-like redundant striping with erasure coding across multiple cloud storage providers. Erasure coding can also be applied to MetaSync and is part of our future work. SpanStore [77] optimizes storage and computation placement across a set of paid data centers with differing charging models and differing application performance. As they are targeting general-purpose infrastructure like EC2, they assume the ability to run code on the server. BoxLeech [33] argues that aggregating cloud services might abuse them especially given a user may create many free accounts even from one provider, and demonstrates it with a file sharing application. GitTorrent [8] implements a decentralized GitHub hosted on BitTorrent. It uses BitCoin's blockchain as a method of distributed consensus.

Perhaps closest to our intent of MetaSync is DepSky [9]; it proposes a cloud of clouds for secure, byzantine-resilient storage, and it does not require code execution on the servers. However, they assume a more restricted use case. Their basic algorithm assumes at most one concurrent writer. When writers are at the same local network, concurrent writes are

coordinated by an external synchronization service like ZooKeeper. Otherwise, it has a possible extension that can support multiple concurrent updates without an external service, but it requires clock synchronization between clients. MetaSync makes no clock assumptions about clients, it is designed to be efficient in the common case where multiple clients are making simultaneous updates, and it is non-blocking in the presence of either client or server failures. DepSky also only provides strong consistency for individual data objects, while MetaSync provides strong consistency across all files in a repository.

Our implementation integrates and builds on the ideas in many earlier systems. Obviously, we are indebted to earlier work on Paxos [49] and Disk Paxos [29]; we earlier provided a detailed evaluation of these different approaches. Active Disk Paxos [17] improved Disk Paxos protocol by introducing the ranked register built with atomic read-modify-write operations. The number of messages in Active Disk Paxos protocol can be same with one in pPaxos, but Active Disk Paxos cannot be applied to our settings as it requires the read-modify-write operation that is not supported by the existing APIs. We maintain file objects in a manner similar to a distributed version control system like git [30]; the Ori file system [53] takes a similar approach. However, MetaSync can combine or split each file object for more efficient storage and retrieval. Content-based addressing has been used in many file systems [13, 19, 50, 53, 68]. MetaSync uses content-based addressing for a unique purpose, allowing us to asynchronously uploading or downloading objects to backend services. While algorithms for distributing or replicating objects have also been proposed and explored by past systems [18, 62, 66], the replication system in MetaSync is designed to minimize the cost of reconfiguration to add or subtract a storage service and also to respect the diverse space restrictions of multiple backends.

Chapter 3

IPV6 PSEUDONYMS

This chapter discusses another challenge: linkability of user behavior. We take the approach of designing a cross-layer architecture to provide users with control over how user behavior can be linked across layers. This work was first described in a 2013 paper [35].

In particular for the network layer, we leverage the size of the IPv6 address space. Unlike today's world, where IP addresses are limited and thus, every activity of every user on a machine (or every machine behind a NAT) is lumped into the same abstraction of a single machine, IPv6 addresses are able to provide a different grouping. The larger address space of IPv6 can potentially allow every activity of every user to have a different address (in fact, there are more than half a million addresses per square nanometer of the Earth's surface).

However, it is not enough to give each user a pile of IPv6 addresses if the entire pile can be traced back to the user. Pseudonyms also need to encompass more than just an IP address as activities can be linked using traditional application- and system-level information. For example, in the case of browsers, potential sources of linkability can be found in cookies, HTML5 LocalStorage, Flash LSOs, and implicit information like user-agent and system fonts [78]. Our system ensures that all of these elements are consistent within a pseudonym, and privacy-preserving across pseudonyms.

To the best of our knowledge, this system was the first to provide integrated cross-layer control over privacy and present it as a first-class abstraction. To this end, we develop a system that allows for each machine to have numerous pseudonyms and design a complementary network layer that can handle a large number of seemingly random, flat addresses with no path dilation, no increase in routing table size, and negligible performance decrease, all while maintaining network transparency. We also delve into a case study of browsers, where

we explore issues related to the isolation of pseudonym information, and policies aggregating activities into pseudonyms. Our simple policy system allows us to implement a variety of protection mechanisms, each of which has different functionality-privacy tradeoffs. Finally, we have implemented a Chrome extension and cloud gateway service that approximates a pseudonymous browser in today's Internet (i.e., an IPv4 Internet without network/OS support).

3.1 Motivation

Advertisers and web services have strong incentives to track the behavior of users on the Internet—often without user consent, control, or knowledge. A recent study conducted by Roesner et al. [70] shows the prevalence of third-party trackers across both popular and randomly sampled domains.

Tracking can sometimes be beneficial to all parties involved, by providing customization and security along with a revenue model for many web services. On the other hand, tracking can just as easily be abused. For example, they might correlate the collected traces to other sources of users' personal information, such as gender, zip code, name, or sensitive search terms [46] to obtain detailed profiles. These profiles can then be used to perform price discrimination [55] or be used by advertisers to display behavior-based or profile-based ads, which sometimes reflect sensitive interests (e.g., medical issues or sexual orientation) [76].

In this section, we first outline the techniques employed by services to track users and show tracker prevalence on the web. Then we discuss why existing solutions such as NATs and anonymizing proxies are not sufficient to support the broad range of policies that a user might desire.

3.1.1 Tracking Methods

The tracking mechanisms available to webservices/advertisers are numerous and span the entire network/application stack. These mechanisms are often out of the control of the user, or otherwise difficult and inconvenient to spoof. They include:

- **IP address:** IP addresses let adversaries correlate user activity without requiring additional application-level information. Even when the IP address changes, as long as other information persists across the change, the new IP address can be linked to the user. It is relatively easy for services to link activities generated by the same IP.
- **Application information:** Often, the application itself will leak information. Web browsers, for example, have cookies, Flash LSOs, and HTML5 LocalStorage that can be used to track the behavior of users.
- **DNS:** The address of the server accessed by the user can reveal information as well. CDNs, for example, may use DNS to distribute requests, and if these entries are reused, subsequent accesses can be linked to the original.
- **System information:** These include time zone, screen size, system fonts, etc. They can be mined through applications. These values are often important for proper functionality, but studies have shown that the set of values is often unique across different users [25].

3.1.2 Tracker Prevalence

As reported in previous studies, third-party trackers are prevalent on the web. We conducted small-scale measurements by using web usage traces of authors and collaborators to see how often we encountered third-party trackers while webbrowsing. The dataset covers three days of web traces from eight users. The traces were generated via a Chrome web browser extension developed by the authors that logs details of each HTTP request made by the browser. Those details include information such as the URL accessed, the HTTP header, an identifier for the browser tab that made the request, a timestamp, and what caused the request (e.g., a recursive request for some embedded object or a result of the user clicking a link). Figure 3.1 shows the frequency of encounters with the top 20 third-party trackers within our traces[1]. The users visited 406 unique domains during the trace collection. Out of the 406 domains, 281 (69.2%) domains contained at least one third-party tracker.

[1]We used the list of trackers from [70].

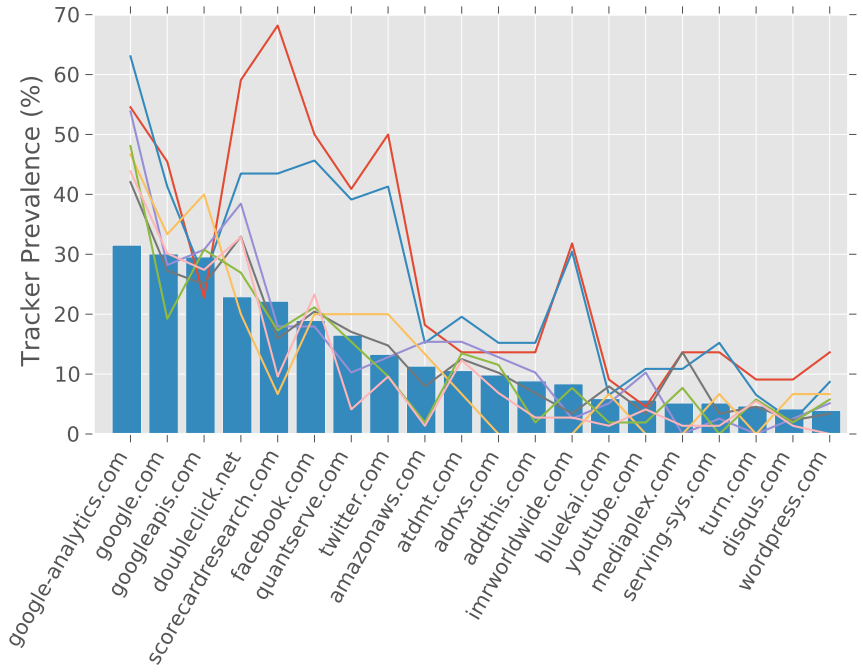

Figure 3.1: **Prevalence of encounters with the top 20 third-party trackers.** We counted encounters within a total number of 406 distinct domains visited over all traces. Each line represents a separate user's trace.

3.1.3 Current Defenses

Most web browsers expose cookie management policies to users (e.g., blocking third-party cookies or only allowing session cookies). Browser add-ons can add finer-grained policies, but regardless of the expressiveness or the flexibility, these application-level solutions do not handle lower-level information like IP addresses.

Network-layer solutions like Tor, proxies, and NATs attempt to alleviate this problem, but none of these are ideal. The default policy on the Internet is that all of a user's behavior can be tracked regardless of what they do, and while these systems provide an alternative policy, the one they enable lacks flexibility: they make IP addresses meaningless. For example, a user might want to allow Amazon to track searches/purchases in order to receive better

recommendations, but at the same time may not want to allow trackers like DoubleClick to correlate interactions across websites. In many cases, there is a fundamental tradeoff between functionality and privacy, and the above solutions do not provide users with control over this tradeoff.

In addition to a lack of flexibility, Tor sacrifices performance because it is an overlay with an attack model that is stronger than what is necessary for an anti-tracking system. Proxies, to a lesser extent, also have performance issues because of path dilation and bandwidth restrictions. NATs may not incur the same penalties, but instead destroy network transparency and violate key tenets of Internet design [15].

The other issue with proxies/NATs is that websites already use IP addresses as hints to increase security[2]. In our system, we attempt to allow applications to leverage IP addresses as a powerful tool for accountability.

3.2 Approach

We allow users to manage a large (potentially unlimited) number of **unlinkable pseudonyms** so that they can choose which pseudonyms are observed by which remote servers. In this case, a pseudonym refers to a set of identifying features that persist across an *activity*—the definition of activity here is dependent on the user's policy.

3.2.1 Threat Model

The fundamental goal of our system is to prevent remote services, with which a user interacts, from linking the user's activities except in ways that the user intends. The adversaries that work against the user and our system are remote trackers that wish to build a more complete profile of that person and/or to perform service discrimination. Their objective is to correlate any and all behavior of each user in an effort to link the user's pseudonyms. To do so, they may also collude with endhosts owned by users in our system to find out pseudonym mappings

[2] `https://support.google.com/accounts/answer/1144110?hl=en`

	Persistent ID	**Transient ID**
Self-contained	Banking Websites	Blogs
Pervasive	Social Media Sites	Analytics

Table 3.1: **Examples of potential adversaries.**

of other hosts in the system.

For web browsing, adversaries may use any information revealed in the packet including, but not limited to, cookies, local storage, fingerprinting, and unique URLs. They may also collude, span multiple services and include other types of entities like CDNs and DNS servers. Their ability to link two activities can be sometimes be probabilistic, particularly in the case of browser fingerprinting, but as soon as an adversary sees sufficient evidence that two activities are from the same user, they, and any associated activities, are thereafter tainted.

Within this framework, websites have varying levels of tracking potential and requirements for functionality. On one axis, there are some trackers that are pervasive and some that only track first-party accesses, and along this axis, services can have differing levels of prevalence (e.g., Google Analytics has a presence on 297 of the top 500 domains, while bluekai has presence on 27 [70]). On the other axis, services require varying levels of persistent identifying information to function. This sometimes depends on the expectations of the user in question, and can range from sites that require login, to sites with recommendation systems, to sites that do not utilize persistent identifiers for functionality and where each access is essentially independent. (See Table 3.1 for examples of this categorization.)

Note that our model does not include intermediaries like routers and ISPs, especially users need to trust their first-hop ISPs. LAP [40] assumes similar threat model for users who want to have intermediate privacy level without sacrificing latencies. In order to defend against these types of attackers including the first-hop ISPs, we would need to launder packets

through other machines, such as in Tor [23] or Herbivore [31].

3.2.2 Pseudonyms

We take the approach that a pseudonym is, to the outside world, the abstraction of a single machine and, to the user, simply a collection of activities that they wish to correlate. Tor, proxies, and NATs, fail to provide this abstraction, and we lose not only the user's control over what activities are correlated, but the ability to use IPs as a single machine identifier as they were intended to be. While one might argue that this removal of IPs as a useful identifier provides more anonymity, *what we are trying to provide is more control, and more control is not synonymous with more anonymity.*

In our approach, a single user may have multiple pseudonyms and may appear to be multiple separate machines/users. This is somewhat similar to Chrome's notion of profiles except that linkability (intra-pseudonym) and unlinkability (inter-pseudonym) need to extend across multiple layers, not just the application-layer. Pseudonyms of the same user should not have information that is both common between each other and unique among all pseudonyms in the system. In general, we either want identifying information to be unique or shared among the pseudonyms of a large set of users across the entire system—the bigger the set, the better.

Pseudonyms should also provide consistent information across the end-to-end connection, and thus maintain network transparency. This prevents accidental leakage of information within the contents of a packet and strengthens the end-to-end principle, avoiding problems endemic to NATs (such as the difficulty of setting up peer-to-peer connections).

3.2.3 Desirable Policies

With pseudonyms, our system is able to concurrently support a variety of potential policies. Below we list a few of the use cases that illustrate the power of our approach.

Sort by Identity: Users use their computers for a variety of different purposes and for a variety of different activities. It is sometimes incorrect to relate these activities, particularly if the computer is used by multiple users or for varied interests and activities. For example, a user may want to create separate pseudonyms for her work-related behavior and what she does in her free time.

Banking Websites: Banking websites track users in order to combat fraud. A common security feature of online banking makes the observation that it is much more likely for access from a new computer to be fraudulent than for accesses from a computer that the user has used in the past. For this purpose, a user may want to use a single pseudonym for all visits to her bank, to convince the banking website that she is the same person who used it previously.

Separate Sessions: Unrelated requests for particular services or resources may need to remain private. For example, different files downloaded by a single user from BitTorrent need not be linked together and could each be in a separate pseudonym without significantly affecting usability of the system. Note that, in this particular example, the pseudonym crosses applications as an adversarial BitTorrent tracker can potentially link the BitTorrent activity to the HTTP download of the .torrent file (and thus the web history of the user).

Block Third-Party Tracking: Even if a user is fine with giving out information to a website to maintain customization or provide analytics, third parties can aggregate information to create comprehensive profiles for individual users. Blocking third party cookies is not sufficient to protect against this because of the information listed in §3.1.1. Stronger protection can be obtained by using a separate, random pseudonym for requests to third-party sites.

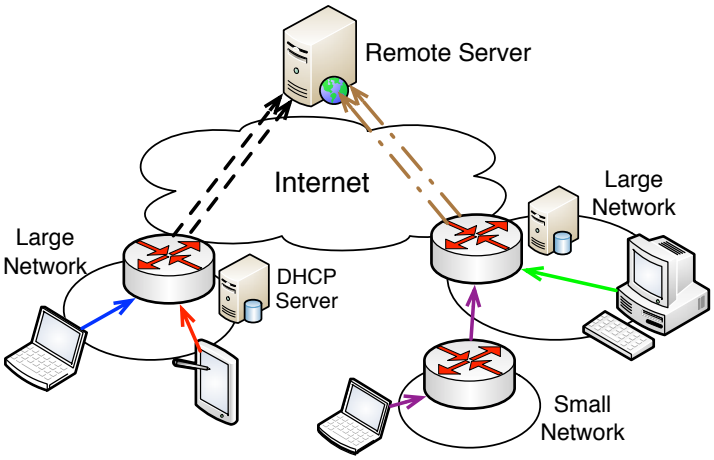

Figure 3.2: **Overview of the network architecture with IPv6 pseudonym.** Color of the lines indicates identifiability—the same colored lines are not discernible.

3.3 System Design

We now outline our design and the changes we make to the network, operating system, and application layers. The pseudonym abstraction necessarily needs to be implemented across many layers—specifically, adversarial web servers can potentially use any information in the network layer and up in order to link pseudonyms and track users. Each layer needs to change to accommodate the ability to allocate/deallocate addresses, and a leak at any layer can potentially break the illusion of separate machines.

Beyond this, our system needs to have answers for a few key challenges:

- What do we need to tackle at each of the layers (network, system, application)?
- What network or operating system support is necessary to handle numerous IPs for each machine?
- How are packets classified into activities or pseudonyms and what potential polices are possible or useful?

At a high level, our system modifies client machines and applications to allow for usage of

pseudonyms, and pseudonyms are mixed with those of other users in the same network/ISP or partnering networks. Networks are responsible for providing IP mixing for their customers, and DHCP servers are responsible for allocating multiple IP addresses to the client upon request (i.e., provide an IP address per pseudonym). Only clients, their applications, and routers within participating networks are modified in our system. See Figure 3.2 for an overview of our system.

3.3.1 Network Layer

In this subsection, we introduce the addressing and routing schemes we use in the network layer to enable unlinkable pseudonyms. The basic goal is to provide each host a large number of IP addresses that can be tied to pseudonyms, and as a consequence, the network layer design should meet the following three requirements:

R1 Proper mixing: The externally visible addresses of pseudonyms corresponding to a host should appear to be randomly chosen. This will prevent destinations and routers external to the mixing ISP from being able to link the activities associated with a host.

R2 Efficient routing: Addressing and routing within the ISP should be efficient. For example, the size of routing tables with the addition of pseudonyms should be comparable to that of routing tables without them, and the per-packet overheads should be minimal.

R3 Easy revocation: The network should be able to periodically remap the association of pseudonyms to externally visible addresses in order to minimize the risks associated with brute force attacks and compromised keys.

Traditionally, addresses in the Internet consist of a network prefix and a host identifier, which can be further divided into a subnet number, and a host number. Routers in the Internet then use longest-prefix matching in order to efficiently route over such addresses. However, if each subnet and computer has a preset prefix, this would leak information to

adversaries about where a particular IP address comes from, and the scheme would fail to meet the requirement R1. We introduce an addressing method where the non-network-prefix portion of addresses *appear* to be randomly chosen from a flat address space, but are still efficiently routable (i.e., can be done at line speed with the same number of routing table entries).

Large Networks: We first consider large networks, where there are enough users to achieve unlinkability by just mixing within the network. In this case, the portion of the address not included in the network prefix should appear to be completely random so as to not leak any information about the subnet or host.

A naïve approach would be to associate each pseudonym with a randomly generated IP addresses in the network and route based on IP addresses within the ISP. However, this approach requires routers to maintain routing tables that are proportional to the number of actively used pseudonyms, with the random address allocation preventing the use of aggregation of routing table entries. Such a scheme would fail to meet R2.

We instead leverage symmetric key encryption to encode the host identifier in a way that is not discernible to outside adversaries and yet can be efficiently routable inside the ISP. At a high level, addresses are constructed using an initial base address consisting of a network prefix, subnet identifier, host identifier, and pseudonym number, with the first three values representing exactly what they do in today's systems. The addition we make is to give each host multiple pseudonym IDs in order to generate distinct addresses for each pseudonym.

The base address is used to generate an **encrypted address** that obfuscates identifying information and will appear to be drawn randomly from a flat address space. More specifically, the portion of the address that includes the subnet ID, host ID, and pseudonym ID is encrypted using a symmetric key as shown in Figure 3.3. Assuming the network has a /64 prefix, the later part of the base address is encrypted with 64-bit symmetric key block encryption[3]. The key is only known by the routers and DHCP servers, as are all pieces of

[3]The recommended allocation for end sites is at least one /64 block [60]. In the case of larger blocks, we

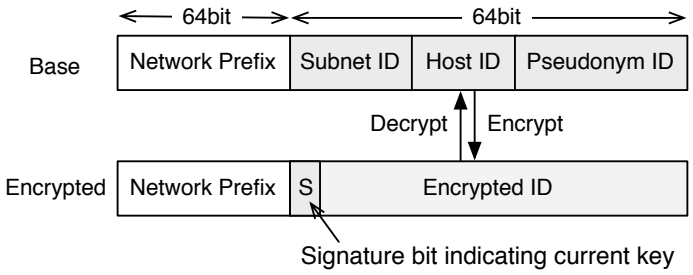

Figure 3.3: **Translation of a base address and encrypted.** Note that some of this address space may be reserved for a DMZ. The base address is not known to anyone except the mixing network's routers and DHCP servers; the outside world and end host only see the encrypted address.

the base address to protect against known-plaintext attacks.

On each incoming packet, routers within the network perform a single symmetric key decryption on the destination address to uncover the base address, which they can then use to forward packets efficiently. Base addresses can be longest-prefix routed and aggregated on subnets or hosts, exactly as in the current Internet. We therefore do not increase routing-table state compared to current IPv6 networks. Also note that we maintain network transparency as routers do not overwrite the IP address, and the end host sees the same encrypted address as the other end of the connection.

Guessing the symmetric key with a brute-force attack is computationally very difficult; while a host knows that the encrypted addresses provided to it are obtained from base addresses that have a common set of bits, exploring the entire key space to identify the secret symmetric key is computationally challenging (e.g., our prototype uses TDES for encryption, where the key space has 2^{168} keys). However, best practices for key management require ISPs to periodically change the encryption keys. To allow easy revocation (R3),

---

can simply ignore the higher bits or segment the users across multiple /64 blocks.

networks can create a new key while allowing users to phase out old keys by appropriating a *signature* bit at the beginning of the encrypted ID and requiring that all encrypted addresses that use a particular key have that bit set at a specific value—there can only be two active keys at any given time and all addresses without the correct bit for the current key will be thrown out. This may decrease the available address space unevenly for particular hosts, but the average should simply be a factor of 2. Each router maintains up to two symmetric keys (the active key and possibly the old key) and determines which key to use based on the initial bit of the encrypted ID.

When a machine wishes to allocate IP addresses, it broadcasts a DHCP request along with its hardware address and the number of desired pseudonym IDs. The DHCP server will generate the requested number of random pseudonym IDs, and use them to generate a set of base addresses. It will then encrypt them with the active secret key and then send them back to the host. If any of the addresses do not have the correct signature bit for the current key, they will be thrown out and replaced before being sent to the host. This increases the work done by the DHCP server by a factor of two, which we believe is acceptable overhead.

It is also possible to design almost stateless DHCP in this system. If the conversion between hardware address to host ID is consistent (e.g., with a consist hash function), then the DHCP server does not need to store any mappings, it simply needs to know the encryption key and hash function and can generate pseudonym IDs on-the-fly. Any machine with those two pieces of information can serve as a DHCP server. It is possible that this technique can result in redundant pseudonym IDs, but the host is required to maintain the set of encrypted addresses currently being used by all of its applications and will filter out duplicates before passing on the DHCP results to the requesting application. Note that the same pseudonym ID should not be used twice for the same host and key in order to prevent unintentional linking of activity. If we run out of pseudonym IDs, either the network administrator should have allocated a larger portion of the address for the pseudonym ID or the host should acquire a new host ID.

Lastly, we allow for truly static IP addresses within a network (e.g., for web servers)

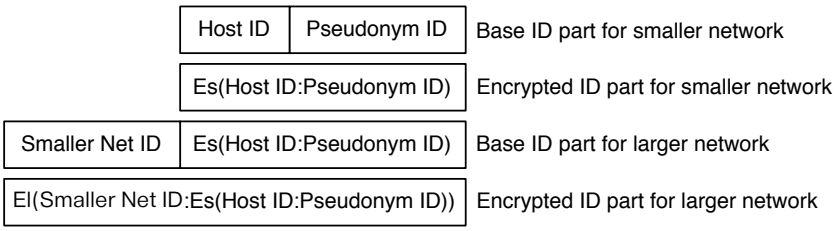

Figure 3.4: **Hierarchy of encrypted addresses for smaller and larger networks.** Note that prefix is not shown here. The final encrypted ID is prefixed with the larger network's prefix.

by reserving prefixes for DMZ IP addresses. For example, the network can specify that all addresses starting with `0000` or `1000` be reserved for the DMZ. Such addresses are not encrypted, and like the 1-bit signature, the DHCP server will throw out and regenerate any encrypted addresses that overlap with this range. The length of these reserved prefixes is variable and so is the amount of extra work done by the DHCP server.

Smaller Networks: For smaller networks that do not have enough users to hide individuals effectively, one solution is to merge their address pools and delegate address assignments to a larger, possibly adjacent, network. Allocation requests from hosts in the smaller network will be forwarded to the larger network's DHCP server. Because there is no mutual trust among the two networks, we add two more requirements for this case:

R4 The larger network (L) should not learn about or control the mixing of the smaller network's (S's) hosts.

R5 S should not learn L's encryption key.

To fulfill these requirements, address allocation is done hierarchically as shown in Figure 3.4. Upon receiving a DHCP request from a host, S generates psuedonym IDs for the host. Then, it encrypts ⟨hostID:pseudonymID⟩ with a smaller block encryption cipher. The

encrypted address is forwarded to L, and L attaches it to S's ID assigned by L, and encrypts it again with L's encryption method. Finally, this resulting address is used along with L's network prefix as the externally visible IP address for the endhost.

Since this address has L's network prefix, incoming packets will be first routed to L. Upon receiving a packet, L decrypts the address with its own key and finds out that it needs to send the packet to S. Since S might not be adjacent to L, L encapsulates the incoming packet with an address of Prefix(S):$E_s\langle$HostID:PseudonymID$\rangle$ and routes it towards S's ingress routers. Upon receving this packet, S can locate the final destination within its network by decrypting the address stored in the encapsulation header with its key. The encapsulation is stripped off just before the packet is delivered to the endhost. Note that potential MTU violation may lead fragmentation, but it could be made to work reliably in IPv6 [71].

As HostID is encrypted together with pseudonym ID before a packet is sent to L, L cannot determine whether two IDs are allocated to a single host. Also, L's encryption key is not exposed to S. Thus, this mechanism meets R4 and R5. Support for key revocation and DMZ can be achieved just as with larger networks (as discussed earlier).

Deployability: The proposed architecture above does not require changes to the IP address format or inter-domain routing. Thus, it is partially deployable (i.e., only participating networks can adopt the above addressing and routing mechanisms without affecting other networks). While the design described in this section provides an appropriate long term solution, one which allows network operators to natively contribute to greater privacy, it requires changes to an ISP's network components (e.g., DHCP servers and routers).

To ease adoption without requiring changes to all routers in the network, translators can be deployed in the network at the appropriate choke points through which all traffic passes. Before packets are introduced into the network, the translator performs a decryption and rewrites the IP address. Additionally, just before a packet arrives at the end host in the network or just before the packet is delivered to the application at the end host, another translator re-encrypts the address, thus avoiding packet header decryption at every hop inside

the ISP. Modifications to DHCP servers are still required in order to provide end hosts with encrypted addresses, but these modifications are relatively minor.

Moreover, we can also use overlay solutions to traverse portions of the Internet that do not support IPv6. These include tunneling over an IPv6 broker and application-layer proxies (e.g., HTTP proxies) We will describe the design and implementation of one such proxy and the corresponding browser extension in §3.5.

3.3.2 Operating System Layer

The operating system maintains the set of addresses and acts as a mediator between the DHCP server and the application. The number of addresses maintained is a preconfigured value that depends on the memory and processing power limitations of the machine in question. That being said, microbenchmarks have shown that, with minor modifications, modern operating systems can handle thousands of addresses with negligible penalties (see §3.6.1).

The OS, like the application and network layer, can also be a source of linking information. DNS lookups and caching are an example of this problem, and for this reason, pseudonyms do not share DNS caches. CNN can point a user to a particular Akamai server and set a long TTL, but it will only be cached for that particular pseudonym. Lookups should be done using a DNS resolver that could be accessed by any node in the mixing network. For example, if Comcast is mixing addresses for its customers, then (a) customers should be able to use any Comcast DNS resolver, (b) the resolvers should be indistinguishable, or (c) the customers should use a third-party resolver (e.g., Google, OpenDNS).

3.3.3 Application Layer

Each application is responsible for setting policies and associating requests to pseudonyms. We dive into the policies and application-level concerns for one very common application in §3.4: web browsers. The policies of other applications as well as the manner in which they protect against information leakage are beyond the scope of this work.

```
1: n ← 100
2: pseudonyms ← ALLOC_PSEUDONYM(n)
3: BZERO(&hints, sizeof(hints))
4: hints.ai_family ← AF_INET6
5: hints.ai_socktype ← SOCK_STREAM
6: hints.ai_protocol ← IPPROTO_TCP
7: GETADDRINFO("www.google.com",80, &hints, &dest)
8: for (i = 0; i < n; ++i) do
9:     sock ← SOCKET(INET6, SOCK_STREAM, PROTO_TCP)
10:    BIND(sock, &pseudonyms[i], sizeof(pseudonyms[i]))
11:    CONNECT(sock, dest → ai_addr, dest → ai_addrlen)
12:    SEND(sock, data, dataLen, 0)
13:    CLOSE(sock)
```

Figure 3.5: **Example code using pseudonyms.** It allocates 100 pseudonyms and uses them to send data using 100 different IP addresses

The interface we provide to applications is simply a modification to the socket interface. We add a function called `allocpseudonym()` that takes as a parameter an integer representing the number of desired pseudonyms. The function returns an array of `sockaddr_in6` structures with information about each pseudonym's address. Similarly, we also provide `freepseudonym()` that applications use to release an address back into the pool of available pseudonyms. IP addresses can be leased/refreshed for a period of time, but there is maximum number of held addresses for both machines and applications. An example usage is listed in Figure 3.5.

A default pseudonym exists for backward compatibility and ease of programming for applications that do not require unlinkabililty. Also note that multiple applications can use the same pseudonym if they wish to associate with each other. For example, a stand-alone banking application may wish to associate itself with the pseudonym that the customer uses

Protection Mechanism	Policy	Pseudonym ID to Use
Trivial	Every request uses the same pseudonym	default
3rd-party blocking	For 3rd-party request, use random pseudonym	3rd-party?requestId:default
Per browsing session: opening a new tab or typing new URL makes a new session	Pseudonym per tab and new page (no referer)	tabId.(sum += referer?0:1)
Per 1st-party (Milk [74]): force 3rd-party to set different cookies for each 1st-party	Pseudonym per 1st-party domain	domain(tabUrl)
Per tab (CookiePie [2]): Use different cookies for each tab	Pseudonym per tab	tabId
Per page: Use different cookies for each page	Pseudonym per page	tabUrl
Private browsing	New pseudonym for private browsing window	private?windowId:default
Per request: No activity should be correlated	Pseudonym per request	requestId
Time-based (TorButton [67])	Allocate a new psuedonym every 10 minutes	Time.Now / 10 minutes

Table 3.2: **Example pseudonym policies.** Our framework could implement most of protection mechanisms proposed by prior work as policies. 1st column: name and description for the privacy protection mechanism. 2nd column: implementation with pseudonym. 3rd column: the policy's pseudonym ID mapping.

inside the browser to access its website.

3.4 A Pseudonymous Web Browser

Any Internet communication can potentially be used to track users, but in this paper, we concentrate on a particularly common avenue of attack: web requests from browsers. For every application, we need to answer two questions: (a) what is contained within a pseudonym and (b) how is traffic classified into pseudonyms. Below, we discuss these issues in the context of web browsers.

Components: Within a web browser, tracking of users is often done through explicit mechanisms like browser cookies, Flash LSOs, or other forms of local storage [70]. Thus,

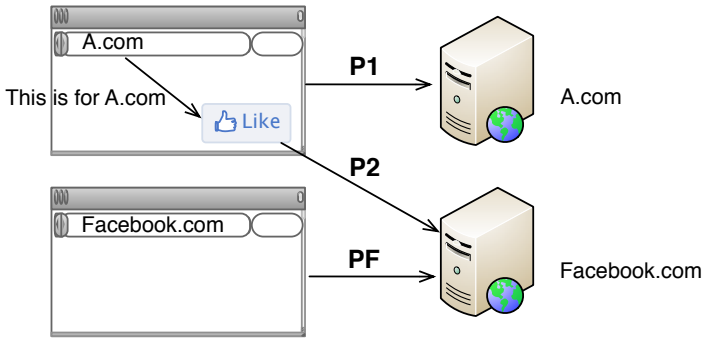

Figure 3.6: **An example of pseudonym policies.** Here, the website A.com has a Facebook widget for registering likes. A web request to A.com results in a third-party web access to Facebook, which would contain information regarding the original page.

each pseudonym needs to have separate storage for these entities in addition to using separate IP addresses. There is also plenty of implicit information that can be used to correlate user behavior [25, 78], and the browser must manage all of these pieces of information as part of a pseudonym.

Policies: Recall that a pseudonym represents a set of linkable activities. It is possible to provide users with the ability to explicitly specify a pseudonym for each web request manually; however, this quickly becomes infeasible. Therefore, we provide policies to allow the system to assign these pseudonyms automatically.

To a first approximation, there are two properties that characterize the usefulness of a particular policy: website functionality and user privacy. These properties are subjective and sometimes at odds with each other. While most users probably expect logins to still work, some may be willing to trade recommendation system accuracy or targeted ads for more privacy, and some might not. In our system, we intend to provide a policy framework that is flexible and expressive enough to allow for a broad range of policies.

To illustrate how a typical policy might work, Figure 3.6 shows an example scenario

where the Facebook social widget is embedded on a web page, A.com, and each of the three resulting requests is assigned some pseudonym (i.e., P1, P2, or PF). Without any privacy protection mechanism, pseudonyms P1, P2 and PF are the same. The other extreme is that every request uses a different pseudonym (P1 $\neq$ P2 $\neq$ PF). We can also imagine a policy that blocks third-party tracking (P1 = PF $\neq$ P2) and a policy that assigns a pseudonym per first-party site (P1 = P2 $\neq$ PF). In the latter three cases, Facebook cannot track the user's visits to `A.com`, but they do not allow Facebook's Like button to function either. Note that in the remaining case (P1 $\neq$ P2 = PF), Facebook can track the user, as it knows the Like button request is coming from `A.com`. Each policy exhibits a different point in the functionality-privacy trade-off, although some combinations (e.g., P1 = PF $\neq$ P2 by default, but link PF and P2 when the user explicitly intends it, as shown in ShareMeNot [70]) allow users to have both.

In our system, pseudonyms are assigned to requests based on information about the request as well as the state of the browser (e.g., unique tab ID, tab URL, unique request ID, whether private browsing mode was enabled, whether the request targets a third party website, etc.). Specifically, policies define a mapping from requests to pseudonym IDs, where IDs are unique, arbitrary strings of characters. A pseudonym per-tab policy, for instance, might simply use the tab ID as the pseudonym ID. Similarly, a Chrome-like private browsing policy might use the window ID of the request as the pseudonym ID for private requests, and use a default pseudonym ID (see §3.3.3) for all other requests.

This policy framework is expressive enough to implement many of the recent privacy protection mechanisms that have been proposed to defend against web tracking. Most of them focus on separating browser cookies or blocking specific requests, but we are able to extend these mechanisms into pseudonym policies, which cover both IP addresses and cookie stores. Table 3.2 shows how several privacy protection mechanisms can be implemented via pseudonym policies.

Users may also combine policies (e.g., a per-tab policy with third-party blocking). More complex policies, such as one that emulates ShareMeNot, can be implemented with a toggle

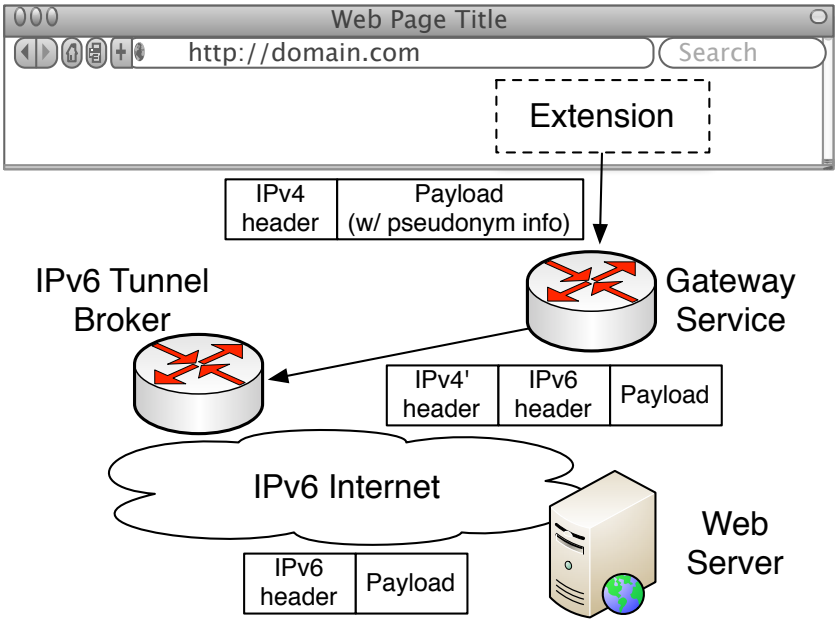

Figure 3.7: **Overview of the Implementation.**

button indicating whether third-party authentication is allowed (if allowed, we use the default pseudonym, otherwise a random ID is chosen). Even more complex policies involving arbitrary relationships, like search term keyword clustering, can be implemented as well, but the point is that our system is powerful enough and expressive enough to support a vast array of policies.

3.5 Implementation for Web Browsers

We have implemented an approximation of our ideal design that enables pseudonyms for web browsing in today's Internet. We are able to provide pseudonyms without modification to the browser, operating system, or network. Ideally, these layers would have support for pseudonym allocation and usage, and IPv6 would be available without tunneling through a broker, but our system provides a proof of concept and path to partial adoption. Note that IP address separation across pseudonyms only works when the destination server is using

IPv6 addresses; however, cookie separation works even with IPv4 servers.

Our prototype implementation consists of two main components: a gateway service that allocates/translates IPv6 addresses and a browser extension that controls pseudonyms and their usage. The gateway service is implemented as a stand-alone proxy running on Linux, and the browser extension is made for the Chrome web browser. The overall architecture is illustrated in Figure 3.7. These components work together to allow each pseudonym to have its own set of cross-layer features that include:

- **IP addresses:** The external IP address for each connection is explicitly indicated by the extension in the header of the HTTP request. The extension allocates/deallocates IPv6 addresses by communicating with the gateway.
- **Name resolution:** Requests for web pages arrive at the gateway with domain names rather than IPs. Name resolution takes place on the gateway instead of the local machine so DNS mappings cannot be used to track hosts.
- **Cookie/LocalStorage:** Sets and gets of local storage are intercepted and modified by the extension so that cookies for separate pseudonyms are isolated from each other.
- **Hardware specs, system information and browser details:** Browser details like the `user-agent` or fonts can be manually set to values that are generic so as to not identify the user. Users can tell how unique their browser details are by utilizing services like Panopticlick [25]. Ideally, this tool would be able to suggest values, provide per-gateway statistics, and potentially utilize per-pseudonym value changes, but this is left for future work.

Each of these pieces of identifying information needs to be consistent within a pseudonym, but crafted such that they are unlinkable to each other. Note that we consider higher-level leakage of information such as form data orthogonal.

3.5.1 Utilizing IPv6 Without Modification to the Network

Our implementation is designed to be deployable immediately even though we rely on IPv6 addresses. While actual use of IPv6 is not yet common—only about 14.6% of ASes were running IPv6 when the original paper was written, and now the proportion is up to 21.5%[4]—the support is there in both OSes and many of the larger services. Because of these limitations, traffic from our system is tunneled through an IPv6 tunnel broker called `Hurricane Electric`[5]. This does not affect the unlinkability of any pseudonyms since the tunnel brokers only see that traffic is coming from our gateway servers. Thus, each client effectively has a single **private** address assigned by its ISP, and many **public** IPv6 addresses assigned by our gateway servers. In the future, widespread IPv6 deployment will allow us to circumvent the tunnel broker and potentially increase performance.

3.5.2 Gateway

Our system utilizes a gateway service that handles allocation of addresses and laundering of packets so that individual clients can use our system without network support. Gateways are implemented as transparent HTTP proxies running on Linux machines. Clients can potentially use multiple gateways to send traffic. Each of these gateways controls a /64 subnet of IPv6 addresses assigned through an IPv6 tunnel broker as explained in §3.5.1, allowing us to dole out and mix traffic over an address space of 2^{64} addresses—much larger than the entire IPv4 address space. The gateway has the following responsibilities:

Allocation/Deallocation: Gateways run an HTTP web server to handle IP allocation and deallocation requests that are generated by the browser extension. The addresses are assigned as they are in our ideal design—the public addresses given to each user are efficiently mappable to the private address and vice-versa without having an easily discernible pattern.

---

[4] `http://bgp.he.net/ipv6-progress-report.cgi` as of Oct., 2012 and as of May, 2016

[5] `http://tunnelbroker.net`. There are also other tunnel brokers like `SixXS`, `http://sixxs.net`.

When a client requests an IP allocation, the gateway creates a set of base addresses using its own network prefix and the IPv4 address of the requester as host ID. It then generates encrypted addresses as detailed in §3.3.1 before sending the list of IP addresses back to the client. We implemented Triple DES (TDES) using `OpenSSL` to perform 64-bit block encryption on the non-network-prefix portion of the address.

Forwarding: After allocation is complete, clients are responsible for crafting web requests to include the encrypted IPv6 address that is to be used for that connection, which it does by adding a string to the header's `user-agent` field. When the gateway receives requests, it reads, decrypts, and removes the IPv6 address from the header and verifies that the client is allowed to use the address. If the user's identifier matches, it checks whether the pseudonym is active for the particular user. (The state of pseudonym IDs is stored in a bit vector.) If the address is valid and active, it then sets up a connection to the endpoint with a socket bound to the specified IP address and uses the connection to forward requests.

As described above, web requests arrive at the gateway with a domain name, allowing the gateway to ensure that all hosts use the same DNS resolver and mappings.

3.5.3 Browser Extension

We have built a Chrome browser extension to implement the client-side component of our system, but the same functionality can be implemented in other browsers that have an extension architecture (e.g., Firefox). The architecture and layout of the extension is illustrated in Figure 3.8. The extension is responsible for managing pseudonyms and assigning them to requests according to policies. It does this by separating cookies on a per-pseudonym basis, intercepting every request made from the web browser, and adding a tag to indicate to the gateway which IPv6 address to use. These functionalities are described in more detail below:

Policy Specification: Policies are defined along with a mapping function that is used to define which pseudonym to use for each request. As a starting point, we implemented each of

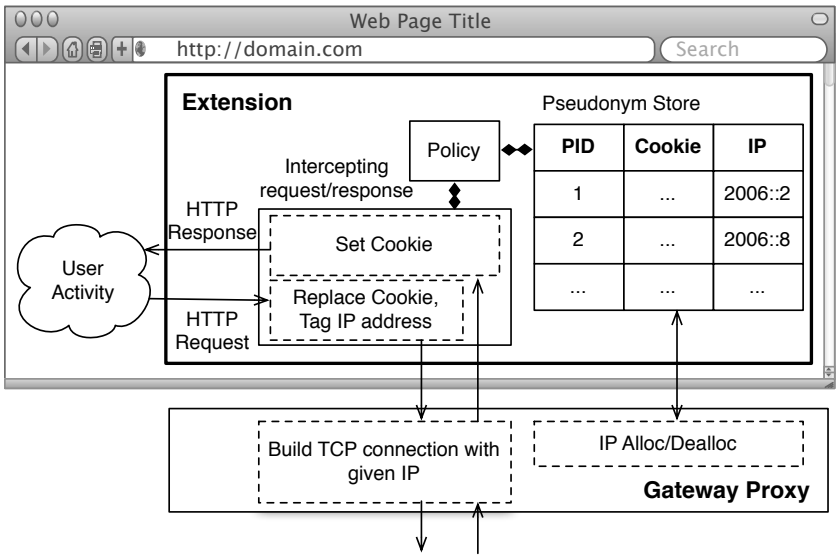

Figure 3.8: **Diagram of the structure of our implementation.** It includes the main components: the extension and gateway proxy.

the policies detailed in Table 3.2, but more sophisticated users can use JavaScript to define their own request to pseudonym ID mappings.

Request Interception: The extension adds listeners for web requests and responses with `chrome.webRequest` APIs (i.e., `onBeforeSendHeaders` and `onHeadersReceived`), which allow it to read and modify HTTP requests and responses. When an HTTP request is captured by the extension, it uses the policy-defined mappings to determine which pseudonym to use for the request. Incoming HTTP responses can be mapped to a specific pseudonym by using a unique identifier that Chrome provides called *request ID*.

Tagging: To notify the gateway of which IPv6 address to use, the extension appends the desired IPv6 address to the `user-agent` passed in the HTTP header, which will be examined and stripped at the gateway. While HTTPS packets are encrypted and therefore cannot be examined at the proxy, the HTTPS tunnel is built using the `CONNECT` protocol. A `CONNECT`

```
1: document.__defineSetter__ ('cookie', set_cookie);
2: document.__defineGetter__ ('cookie', get_cookie);
3: procedure SetCookie(rawCookie)
4:     tuple ← rawCookie.split('=')
5:     remove_cookie(pseudonymID+ '_' +tuple[0])
6:     __cookie += pseudonymID+ '_' +value+ ';'
7: procedure GetCookie
8:     ret ← "", cookies ← __cookie.split(';')
9:     for all c in cookies do
10:        PID ← c.substring(0, c.indexOf('_'))
11:        if PID = pseudonymID then
12:            noPIDcookie ← c.substring(c.indexOf('_')+1)
13:            ret += noPIDcookie+ ';'
14:    return ret
```

Figure 3.9: **Pseudocode for overriding JavaScript cookie code.** A similar technique can be used for HTML5 LocalStorage.

packet includes a `user-agent` field that we can use to communicate pseudonym information[6].

Address Allocation/Deallocation: When the extension is low on IPv6 addresses, it sends an allocation request to the gateway in the form of an HTTP request. It releases these addresses if they are ephemeral and will not be used again. It also maintains a reserve pool of addresses (currently 10) to prevent requests from being blocked due to the lack of free IPs.

Cookie Storage: To separate cookie stores on a per-pseudonym basis, we prepend a pseudonym ID onto each cookie key. This makes every cookie unique to the particular cookie

[6]As of the time of writing, Chromium/Chrome has an issue where it does not expose the CONNECT request to the extension. This has been classified as a bug: `http://code.google.com/p/chromium/issues/detail?id=129572`.

(*path*, *pseudonym*) pair. Cookies are either set through the an HTTP response's `set-cookie` header or using JavaScript. HTTP requests and responses are all intercepted and the cookie-related headers rewritten so that outgoing requests contain the original cookie keys, and incoming responses contain cookie keys that have been prepended with the psudonym ID. Cookie (as well as HTML5 LSO) accesses through JavaScript getters and setters are overridden by custom JavaScript functions (see Figure 3.9).

3.6 Evaluation

In this section, we present the evaluation of the performance and privacy aspects of our system. We seek to answer whether our system scales to the required number of pseudonyms under various privacy policies and how different policies compare with regards to the preservation of privacy. We drive the evaluation of our system by a combination of performance measurement and web-usage trace study, utilizing the HTTP request traces presented in §3.1.2.

3.6.1 Performance

To evaluate the performance implications of our system, we utilize a combination of an end-to-end comparison of page load time and micro benchmarks using our prototype implementation. We find that current technology already has more than enough capacity to implement each aspect of our system.

End-to-End Performance: To evaluate the overall performance of the implementation, we conducted an end-to-end measurement of page load times of the top 100 Alexa[7] websites that support IPv6. Note that 25% of the overall top 100 Alexa websites support IPv6, and that the 100th IPv6 supporting web site was ranked 869th in the whole ranking.

We use the Chromium benchmarking extension[8], which collects HTTP connection per-

[7] `http://www.alexa.com/topsites`

[8] `http://bit.ly/14ErTju`

	User	1	2	3	4	5	6	7	8
Trivial	Pseudonyms	1	1	1	1	1	1	1	1
	Activities	651	4657	359	872	63	692	599	615
	Collections	38	101	338	515	91	336	179	166
Per tab	Pseudonyms	36	123	64	174	36	186	111	412
	Activities	9.50/41	14.15/67	5.02/31	4.48/40	1.75/17	3.48/30	4.33/33	1.21/16
	Collections	0.94/9	0.81/12	5.00/22	2.64/34	2.31/8	1.55/7	1.45/20	0.31/5
Per browsing session	Pseudonyms	140	858	178	350	38	244	227	670
	Activities	4.65/28	5.43/33	2.02/27	2.49/34	1.66/17	2.84/17	2.64/30	0.92/16
	Collections	0.27/9	0.12/12	1.90/10	1.47/13	2.39/5	1.38/6	0.79/9	0.25/5
Per 1st-party	Pseudonyms	425	730	393	655	158	534	579	1319
	Activities	0.04/15	0.03/19	0.04/16	0.05/36	0.03/4	0.04/23	0.03/17	0.00/5
	Collections	0.06/26	0.11/77	0.56/220	0.49/318	0.35/55	0.31/166	0.21/122	0.09/121
Per page	Pseudonyms	1276	7701	915	2152	178	1412	1689	3356
	Activities	0.51/28	0.60/33	0.39/27	0.40/34	0.35/17	0.46/17	0.35/30	0.18/16
	Collections	0.00/0	0.00/0	0.00/0	0.00/0	0.00/0	0.00/0	0.00/0	0.00/0
3rd-party blocking	Pseudonyms	22982	29313	9233	16132	1437	11287	18605	33983
	Activities	0.17/13	0.27/19	0.10/16	0.15/36	0.08/4	0.07/21	0.06/17	0.04/5
	Collections	0.00/24	0.00/73	0.03/216	0.03/307	0.05/54	0.02/158	0.01/116	0.00/114
Per request	Pseudonyms	63062	59045	17586	49252	4906	33644	42645	79321
	Activities	0.00/0	0.00/0	0.00/0	0.00/0	0.00/0	0.00/0	0.00/0	0.00/0
	Collections	0.00/0	0.00/0	0.00/0	0.00/0	0.00/0	0.00/0	0.00/0	0.00/0
Time-based	Pseudonyms	155	109	67	126	153	151	106	350
	Activities	8.34/78	19.57/79	4.58/42	9.38/60	0.38/20	3.89/40	7.37/59	1.40/23
	Collections	0.23/13	0.91/9	4.42/18	3.43/20	0.52/11	1.83/20	1.54/20	0.35/5

Table 3.3: **Trace study results under different privacy policies.** The numbers show average/maximum over all pseudonyms. Pseudonyms shows the number of pseudonyms created. Activities shows the number of observed page views by third-party trackers. Collections shows the number of page views and links followed, as observed by first-party trackers.

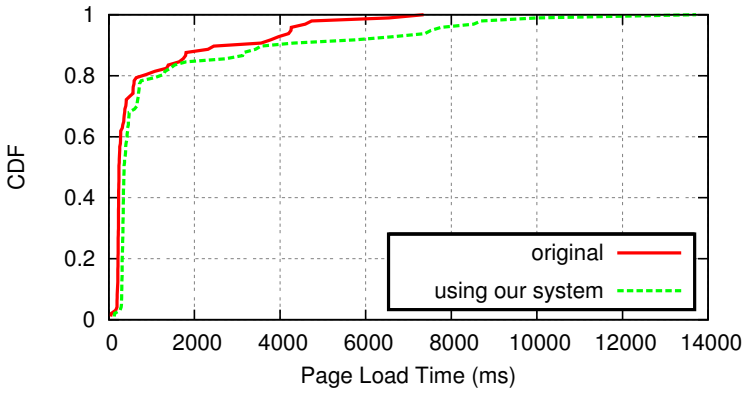

Figure 3.10: **Page load time for Top 100 IPv6 websites.** It show page load time with and without our system.

formance data. We measure web page load times from a desktop Linux machine for each website with and without our system. When using our system, the gateway is located within the local network and the tunnel broker is in the same city. For each iteration, we clear the browser cache and close all open connections. Figure 3.10 shows a CDF of median page load time over 10 iterations of this measurement. We can see that our implementation adds only small overhead. Median and 90th percentile page load time of the top 100 sites increased from 237ms to 367ms, and from 3.6s to 4.3s, respectively.

OS-Level Performance: The number of pseudonyms supported by our system is limited by the number of IP addresses we can assign concurrently to a network interface without performance degradation. For example, the Linux operating system enforces a configurable[9] default limit of 4096 addresses.

We determine the number of pseudonyms that would be required under the different privacy policies and compare this to the maximum number of pseudonyms. The results are shown in the *Pseudonyms* row in Table 3.3 and we see that only in the restrictive cases

[9]Via `/proc/sys/net/ipv6/route/max_size`.

of policies 4 and 5 do we exceed the default limit. The average number of pseudonyms is typically below 1000 for the other policies.

Assigning a large number of addresses to each network interface is feasible today without modification to the Linux kernel. Linux can already assign and utilize 4096 addresses on a single interface with negligible slowdown in performance. There is no reason to believe that a very large number of pseudonyms cannot be used concurrently.

We evaluated the performance of socket operations when a large number of addresses is allocated to a network interface and found no slowdown for most socket operations, including socket creation, connection setup, and sending data (via `connect()`, `accept()`, and `send()`). Binding the socket to an address using the `bind()` system call, however, incurred linear slowdown of up to 8x, depending on the address being bound. Binding addresses took 1 microsecond in the best case and 8 microseconds in the worst case. This is due to the data structure used to store assigned IP addresses. Linux uses a linear list of all IP addresses and a hash of the IP address as an index into the list. The hash only takes the upper 64 bits of an address into account and the list has to be searched linearly for IP addresses that change only the lower 64 bits. With a sufficiently large address pool, traversing the list can incur significant overhead. This can be fixed simply by changing the hash function to incorporate all bits of the IP address. Further tests on an unmodified Linux kernel revealed that OS-level routing tables are not affected by an increase in addresses, and only the data structure that holds the list of valid addresses is affected.

Router Performance: Another potential performance issue is that we add a decryption step for every packet at each router. Although this does increase the complexity of routers, we view this as an acceptable tradeoff for fine-grained privacy control. Recent research shows that encrypting and decrypting entire packets at line speed is possible [42, 45]. Our proposed changes are less extreme in that we only encrypt a portion of the address. From our microbenchmarks, our current TDES implementation decrypts at 2.24 million packets per second with one core of an Intel Xeon E5620 2.40 GHz CPU. This scales to 17.92 million

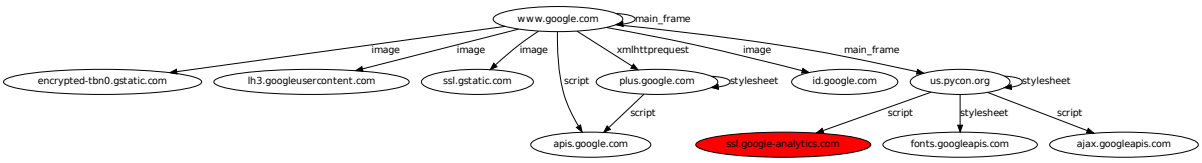

Figure 3.11: **Example collusion graph generated from a single web session.** Nodes represent domains. Edges show which domains caused HTTP requests to which other domains. Edge labels denote the type of request. Red nodes are web trackers.

packets with 8 cores, and assuming 1500 bytes per packet, results in over 200 Gbps.

3.6.2 Privacy Preservation

To investigate how much privacy can be preserved by our system, we analyze how much information a web tracker can collect about each user in the trace study under each of the privacy policies presented in Table 3.2 except d *private browsing*, which was not used by any user in our trace study.

To do so, we use collusion graphs [1] that describe page visits among web domains. Nodes in a collusion graph represent domains and directed edges from node a to node b represent HTTP requests made by the browser to domain b due to an action carried out on a page of domain a, such as following a hyperlink.

Figure 3.11 shows an example collusion graph of a single web session extracted from our traces. It includes a lookup of technical documentation on Google and navigating to a corresponding page (`us.pycon.org`). The page in question uses Google Analytics to track page visits, which accesses several sub-domains to display active content.

Each privacy policy results in a different number of generated pseudonyms. We create a collusion graph for each pseudonym and subsequently evaluate how many different page views a third-party tracker is able to observe. The *Activities* row in Table 3.3 shows this number

for the investigated policies. We see that a separate pseudonym *per browsing session* can drastically reduce the amount of information third-party trackers are able to collect about a user over a vanilla web browser (*trivial*) and that narrowing this policy does little to reduce the amount of information any further (e.g., *per page*). First-party trackers can be limited with a more restrictive policy. A pseudonym *per request*, for example, eliminates all third-party and most first-party tracking, but these policies can also detrimentally impact the browsing experience. The domain-based policy (*per 1st-party*) can provide for a better browsing experience than the policy of a pseudonym *per browsing session* because it allows first-party sites to generate personalized profiles (e.g., a Google search profile).

Anonymizers, such as Tor+Torbutton, which use a timeout-based approach to anonymity and change pseudonyms every 10 minutes (equivalent to our *time-based* policy), are less effective in combating third-party trackers than our simple policy of changing pseudonyms upon every newly entered URL (i.e., per browsing session). However, the difference is small and the amount of information gathered by first-party tracking is almost identical. This is because users typically conduct multiple activities in a relatively short time span on a website, such as entering multiple search terms, before manually navigating on to a new site. In the case of user #5, time-based anonymization is, on average, even more effective than the policy of a pseudonym *per browsing session*. This particular user conducted long web sessions with little activity, such as reading technical manuals and watching online movies.

3.6.3 Summary

Our results show that our system is practical and can be deployed today. Furthermore, it is evident that configurable privacy policies are desirable. Because users have differing expectations for privacy and functionality (e.g., suggestions or directed advertising), there is no one-size-fits-all policy. Even browsing behavior affects this choice, as a slower-paced browsing session can benefit just as much from a timeout-based privacy policy as from a domain-based or a link-based policy. Thus, a system that allows for more privacy control is beneficial.

3.7 Discussion

Without Application Cooperation: For much of this paper, we rely on applications to create and enforce policies for connections that they generate. Unfortunately, legacy applications do not provide this support and may allow tracking. To fix this, we can sandbox the entire application. This sandboxing can be done with minimal OS-level support, or can be done by adding pseudonym support to VM software and running applications within a VM. Alternatively, we can intercept traffic from the application and classify it at a lower level, but this potentially requires application-specific knowledge and cannot handle encrypted communication.

Browser Extension Usability: There are still plenty of improvements we can make to our browser extension that would improve the usability of our system. These might include an indication of what pseudonym is being used for a particular resource or website and a way to change the pseudonym and reload the page. Though a user may have many pseudonyms, we can make management and reuse easier by allowing users to name important pseudonyms or by suggesting pseudonyms if they have previously logged into the website. We can also decrease the need for manual exceptions and unnecessarily restrictive policies with whitelists/blacklists. These concerns are left for future work.

3.8 Related Work

There have been a number of studies that characterize web tracking [47, 48, 70]. Additionally, Eckersley [25], Yen et al. [78], and Nikiforakis et al. [63] point out that implicit information, such as user-agent or system fonts information, along with IP address can effectively fingerprint hosts.

As privacy concerns get more attention, many privacy-protecting mechanisms are being proposed and deployed. Modern browsers have private browsing modes, but these are not very effective at mitigating web tracking [7]. Chrome and Firefox support multiple profiles

to separate identities, but require manual control and still leak implicit information. In addition to these, privacy-protecting extensions have also been proposed. For example, Milk [74] prevents third-party trackers from building user profiles across web sites, and ShareMeNot [70] and Kontaxis et al. [44] provide protection against tracking using social media widgets. The above mechanisms are somewhat effective within their threat models, however, they cannot prevent adversaries from tracking using IP addresses as they are all application-layer solutions.

Also related to this work, there are network- and overlay-layer systems that provide anonymity using onion routing [23, 51]. These proposals are orthogonal to our work as they do not address application-layer privacy and attacks. Our network-layer mechanism provides a subset of the properties provided by these proposals, but is simpler and more efficient. Additionally, our solution focuses on creating unlinkable pseudonyms and giving users more contol over linkability, which is difficult to achieve with Tor as it uses a limited number of exposed IP addresses (one from the current Tor exit node at a given time). Torbutton [67] is a browser extension that addresses application-level security and privacy concerns, such as separating the cookie store between Tor and non-Tor sessions, but provides only limited control over linkability. Privoxy [3] is a web proxy with comprehensive filtering capabilities and can modify web requests based on user configuration. Unfortunately, since it is not attached to the browser, it cannot track activities closely. Moreover, Privoxy cannot handle HTTPS requests as the connections are encrypted. Our system can potentially include most features of Privoxy in our gateway service. The Address Hiding Protocol [69] proposes to assign a random IP address (still from the same ISP) to hide the original source from the destination. RFC 3041 proposes a privacy extension to stateless IPv6 address autoconfiguration that lets hosts generate temporary IPv6 addresses by hashing interface identifiers. These schemes do not address application-layer issues nor do they provide for efficient routing of packets to randomly assigned addresses [59].

Chapter 4

CONCLUSION

In today's Internet, various personal data is easily accessible, which provides much convenience to the users. However, it raises security and privacy risks, as the personal data goes through untrusted or unreliable environments, and users do not have much control over information exposure.

This dissertation explored and tackled two key security and privacy challenges in this untrusted environment: untrusted cloud services and unsolicited linkability of user behavior. To address these challenges, we identified that it is necessary to provide users with more control over how and which information is exposed to the remote services. For the issue of untrusted cloud services, we presented MetaSync (Chapter 2), an efficient and secure file synchronization system across multiple existing untrusted storage services. By combining existing services, and moving the trust to the auditable clients, it could achieve better reliability and performance for file synchronization. Along with the system, we introduced two new algorithms which can be applicable to other domains: a client-based consensus algorithm (pPaxos), and a stable deterministic mapping algorithm. For the issue of linkability, we presented the IPv6 pseudonym abstraction (Chapter 3), which disconnects unnecessarily linked information of a user as much as possible, then gives users flexible control over which activities can be linked together. To support it, we designed a cross-layer architecture that exploits the ample IPv6 address space with an efficient routing protocol.

Together, these two projects provide ways for users to gain more control over how their information is exposed to others without losing much efficiency or degrading performance. As discussed in Chapter 1, this dissertation does not (and cannot) provide a single solution to the issues of maintaining privacy on the Internet, but the presented principles can be

applied more broadly to other problems and platforms.

Concluding this dissertation, we present several potential directions of future research. First of all, each system described in the dissertation can be enhanced by the study of related issues. In MetaSync, we move trust to an auditable client from cloud providers. Even if the client software can be audited, it does not guarantee that it always works correctly. If the client software is verified with software verification techniques [39], it can get more credibility. Verifying systems like MetaSync that interact extensively with external services would pose a new set of challenges. On the other hand, IPv6 pseudonyms allow users flexible control of identities. Giving users more control does not solve all the problems by itself. Without proper guidance to users, it can be deemed useless. Therefore, it is necessary to study how users should use the given choice in the usable security domain [4]. Furthermore, we implemented and studied only known policies in Chapter 3. Given the framework, devising new policies to protect user privacy would be helpful as well. Users may need to select and combine policies according to their browsing patterns. As an example, using natural language processing and machine learning techniques to cluster keywords and to determine proper pseudonyms can be used in the context of web search; a policy may change the pseudonym when querying a potentially sensitive keyword to a search engine.

Next, the new algorithms introduced along with each system—the client-based consensus algorithm, the stable deterministic mapping, and the routing protocol encoding more information in IP addresses—can be further studied and applied to other domains as well. For example, the append-only log abstraction in pPaxos can be used in the design of other consensus algorithms' client-based versions, such as Viewstamped Replication [64] or RAFT [65]. We also briefly discussed how pPaxos can be extended to the case of byzantine fault services in §2.2.6, but it needs actual implementation and complete evaluation to be practical. As sketched in the section, the extension mainly requires cryptographic signature and $3f + 1$ services when f concurrently fail. On the other hand, we can apply similar modification to the BFT algorithm by Castro and Liskov [16]. Instead of multicast through primary, a client is required to broadcast messages and check whether it is prepared and committed-

local at each acceptor, as the multicast cannot be done without modifying APIs from the services. Another important property we can study regarding byzantine fault tolerance is the consistency model of each service. Even if they do not intentionally equivocate the order of messages in the append-only log, a service may only guarantee eventual consistency. It may result in non-deterministic ordering of messages, which can invalidate the assumption in pPaxos. To confirm consistency models empirically, we need extensive testing by sending concurrent messages and checking consistency in the order of messages over time from multiple vantage points.

Finally, emerging platforms and devices pose new challenges and opportunities in security and privacy. One example is a video sensing device, such as wearable cameras and augmented reality devices. These devices may continuously monitor users' surroundings and let applications respond to the sensing inputs based on computer vision recognition tasks. As computer vision tasks are typically compute-heavy and mobile devices are resource-constrained, much computation on the video inputs need to be located in the cloud [38]. With this environment, we will need better privacy protection, as users may not trust the cloud while the video stream contains rich private information. One approach is to process frames locally if the image contains potentially private information, or to split processing between the local device and the cloud in a way that does not expose private information, which requires further studies in computer vision and cryptography.

Another example is an electronic micro-payment system. New forms of decentralized cash, such as Bitcoin [58], have the potential to revolutionize many aspects of electronic payment systems. While Bitcoin has shown that a block chain model can be viable in establishing a banking system without centralized trust, the current design lacks several important properties required to expand its usage (e.g., scalability, short latency, and anonymity guarantees). By applying distributed systems ideas like delegation and sharding to the Bitcoin block chain model, we may be able to create a new micropayment system that can support anonymous low-cost, high volume transactions. Further, many users currently rely on private exchanges for faster transactions and exchanging Bitcoin to the other currencies (e.g.,

dollar), which requires users to put trust into the exchanges. Similar to the issues discussed in MetaSync, these private exchanges may not work correctly as users expect. Instead of blindly trusting them, it would be useful to create an auditable exchange that allows users to validate transactions and to make them irreversible within the exchange service.

BIBLIOGRAPHY

[1] Collusion. http://bit.ly/XZgEM6.

[2] Cookiepie. http://bit.ly/11e3HFE.

[3] Privoxy. http://privoxy.org.

[4] Symposium on usable privacy and security. https://cups.cs.cmu.edu/soups/.

[5] Hussam Abu-Libdeh, Lonnie Princehouse, and Hakim Weatherspoon. RACS: A case for cloud storage diversity. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2010.

[6] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *In Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–14, 2002.

[7] G. Aggrawal, E. Bursztein, C. Jackson, and D. Boneh. An analysis of private browsing modes in modern browsers. In *Proceedings of the 19th USENIX Security Symposium (Security)*, 2010.

[8] Chris Ball. Announcing GitTorrent: A decentralized GitHub. http://blog.printf.net/articles/2015/05/29/announcing-gittorrent-a-decentralized-github/, 2015.

[9] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa.

DepSky: Dependable and secure storage in a cloud-of-clouds. In *Proceedings of the European Conference on Computer Systems (Eurosys)*, pages 31–46, 2011.

[10] Alysson Bessani, Ricardo Mendes, Tiago Oliveira, Nuno Neves, Miguel Correia, Marcelo Pasin, and Paulo Verissimo. SCFS: A shared cloud-backed file system. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 169–180, 2014.

[11] Chad Brooks. Cloud storage often results in data loss. `http://www.businessnewsdaily.com/1543-cloud-data-storage-problems.html`, 2011.

[12] Sean Byrne. Microsoft OneDrive for business modifies files as it syncs. `http://www.myce.com/news/microsoft-onedrive-for-business-modifies-files-as-it-syncs-71168`, 2014.

[13] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows Azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 143–157, 2011.

[14] Canonical Ltd. Ubuntu One: Shutdown notice. `https://one.ubuntu.com/services/shutdown`, 2014.

[15] B. Carpenter. Internet transparency. RFC 2775, 2000.

[16] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)*, 1999.

[17] Gregory Chockler and Dhalia Malkhi. Active disk paxos with infinitely many processes. In *Proceedings of the 21st Symposium on Principles of Distributed Computing (PODC)*, 2002.

[18] Asaf Cidon, Stephen M. Rumble, Ryan Stutsman, Sachin Katti, John Ousterhout, and Mendel Rosenblum. Copysets: Reducing the frequency of data loss in cloud storage. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference (ATC)*, pages 37–48, 2013.

[19] Austin T. Clements, Irfan Ahmad, Murali Vilayannur, and Jinyuan Li. Decentralized deduplication in SAN cluster file systems. In *Proceedings of the 2009 USENIX Conference on Annual Technical Conference (ATC)*, 2009.

[20] Jameks Cook. All the different ways that 'iCloud' naked celebrity photo leak might have happened. `http://www.businessinsider.com/icloud-naked-celebrity-photo-leak-2014-9`, September 2014.

[21] Jan Ćurn. How a bug in Dropbox permanently deleted my 8000 photos. `http://paranoia.dubfire.net/2011/04/how-dropbox-sacrifices-user-privacy-for.html`, July 2014.

[22] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, pages 205–220, 2007.

[23] Roger Dingledine, Nick Mathewson, and Paul Syerson. Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium (Security)*, 2004.

[24] Dropbox API. `https://www.dropbox.com/static/developers/dropbox-python-sdk-1.6-docs/index.html`, April 2014.

[25] Peter Eckersley. How unique is your web browser? In *Proceedings of the 10th Privacy Enhancing Technologies Symposium (PETS)*, 2010.

[26] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *Proceedings of 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.

[27] Ariel J. Feldman, Aaron Blankstein, Michael J. Freedman, and Edward W. Felten. Social networking with Frientegrity: Privacy and integrity with an untrusted provider. In *Proceedings of the 21st USENIX Conference on Security Symposium (Security)*, 2012.

[28] Ariel J. Feldman, William P. Zeller, Michael J. Freedman, and Edward W. Felten. SPORC: Group collaboration using untrusted cloud resources. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2010.

[29] Eli Gafni and Leslie Lamport. Disk paxos. *Distributed Computing*, 16(1):1–20, February 2003.

[30] Git Internals - Git Objects. `http://git-scm.com/book/en/Git-Internals-Git-Objects`.

[31] Sharad Goel, Mark Robson, Milo Polte, and Emin Gun Sirer. Herbivore: A scalable and efficient protocol for anonymous communication. Cornell University Computing and Information Science TR2003-1890, 2003.

[32] Google Drive API. `https://developers.google.com/drive/v2/reference/`, April 2014.

[33] Raúl Gracia-Tinedo, Marc Sánchez Artigas, and Pedro García López. Cloud-as-a-Gift: Effectively exploiting personal cloud free accounts via REST APIs. In *Proceedings of the IEEE 6th International Conference on Cloud Computing (CLOUD)*, 2013.

[34] Seungyeop Han, Jaeyeon Jung, and David Wetherall. A study of third-party tracking by mobile apps in the wild. Technical Report UW-CSE-12-03-01, University of Washington, Department of Computer Science and Engineering, 2012.

[35] Seungyeop Han, Vincent Liu, Qifan Pu, Simon Peter, Thomas Anderson, Arvind Krishnamurthy, and David Wetherall. Expressive privacy control with pseudonyms. In *Proceedings of the ACM SIGCOMM 2013 conference (SIGCOMM)*, 2013.

[36] Seungyeop Han, Haichen Shen, Taesoo Kim, Arvind Krishnamurthy, Thomas Anderson, and David Wetherall. MetaSync: File synchronization across multiple untrusted storage services. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC)*, 2015.

[37] Seungyeop Han, Haichen Shen, Taesoo Kim, Arvind Krishnamurthy, Thomas Anderson, and David Wetherall. MetaSync: Coordinating storage across multiple file synchronization services. *IEEE Internet Computing (IC)*, 20(3):36–44, 2016.

[38] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. MCDNN:An Approximation-Based Execution Framework for Deep Stream Processing Under Resource Constraints. In *Proceedings of the 14th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2016.

[39] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. IronFleet: Proving practical distributed systems correct. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[40] Hsu-Chun Hsiao, Tiffany Hyun-Jin Kim, Adrian Perrig, Akira Yamada, Samuel C. Nelson, Marco Gruteser, and Wei Meng. LAP: Lightweight anonymity and privacy. In *Proceedings of IEEE Symposium on Security and Privacy (Oakland)*, 2012.

[41] Geoffrey Huntley. Dropbox confirms that a bug within selective sync may have caused data loss. `https://news.ycombinator.com/item?id=8440985`, October 2014.

[42] Keon Jang, Sangjin Han, Seungyeop Han, Sue Moon, and KyoungSoo Park. SSLShader: cheap SSL acceleration with commodity processors. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[43] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing (STOC)*, pages 654–663, 1997.

[44] Georgios Kontaxis, Michalis Polychronakis, Angelos D. Keromytis, and Evangelos P. Markatos. Privacy-preserving social plugins. In *Proceedings of the 21st USENIX Security Symposium (Security)*, 2012.

[45] Michael E. Kounavis, Xiaozhu Kang, Ken Grewal, Mathew Eszenyi, Shay Gueron, and David Durham. Encrypting the internet. In *Proceedings of the ACM SIGCOMM 2010 conference (SIGCOMM)*, 2010.

[46] Balachander Krishnamurthy, Konstantin Naryshkin, and Craig E. Wills. Privacy leakage vs. protection measures: the growing disconnect. In *Proceedings of the 7th ACM Workshop on Privacy in the Electronic Society (WPES)*, 2011.

[47] Balachander Krishnamurthy and Craig E. Wills. Generating a privacy footprint on the internet. In *Proceedings of the 2006 Internet Measurement Conference (IMC)*, 2006.

[48] Balachander Krishnamurthy and Craig E. Wills. Privacy diffusion on the web: a longitudinal perspective. In *Proceedings of the 18th International World Wide Web Conference (WWW)*, 2009.

[49] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[50] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–9, 2004.

[51] Vincent Liu, Seungyeop Han, Arvind Krishnamurthy, and Thomas Anderson. Tor instead of IP. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks (HotNets)*, 2011.

[52] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michae Walfish. Depot: Cloud storage with minimal trust. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 1–12, 2010.

[53] Ali José Mashtizadeh, Andrea Bittau, Yifeng Frank Huang, and David Mazières. Replication, history, and grafting in the Ori file system. In *Proceedings of the 24th Symposium on Operating Systems Principles (SOSP)*, pages 151–166, 2013.

[54] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *Proceedings of the 7th Annual International Cryptology Conference (CRYPTO)*, pages 369–378, Santa Barbara, CA, 1987.

[55] Jakub Mikians, László Gyarmati, Vijay Erramilli, and Nikolaos Laoutaris. Detecting price and search discrimination on the Internet. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks (HotNets)*, 2012.

[56] Eric Mill. Dropbox Bug Can Permanently Lose Your Files . `https://konklone.com/post/dropbox-bug-can-permanently-lose-your-files`, 2012.

[57] Martin Mulazzani, Sebastian Schrittwieser, Manuel Leithner, Markus Huber, and Edgar Weippl. Dark clouds on the horizon: Using cloud storage as attack vector and online slack space. In *Proceedings of the 20th USENIX Security Symposium (Security)*, 2011.

[58] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. `http://bitcoin.org/bitcoin.pdf`.

[59] T. Narten, R. Draves, and S. Krishnan. Privacy extensions for stateless address autoconfiguration in IPv6. RFC 4941, 2007.

[60] T. Narten, G. Huston, and L. Roberts. IPv6 address assignment to end sites. RFC 6177, 2011.

[61] Jude Nelson and Larry Peterson. Syndicate: Democratizing cloud storage and caching through service composition. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, pages 46:1–46:2, 2013.

[62] Edmund B. Nightingale, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, and Yutaka Suzue. Flat datacenter storage. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 1–15, 2012.

[63] Nick Nikiforakis, Alexandros Kapravelosy, Wouter Joosen, Christopher Kruegely, Frank Piessens, and Giovanni Vignay. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *Proceedings of IEEE Symposium on Security and Privacy (Oakland)*, 2013.

[64] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 1988.

[65] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC)*, 2014.

[66] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 109–116, 1988.

[67] Mike Perry. Torbutton design documentation. `https://www.torproject.org/torbutton/en/design/index.html.en`, 2011.

[68] Sean Quinlan and Sean Dorward. Venti: A new approach to archival data storage. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST)*, 2002.

[69] Barath Raghavan, Tadayoshi Kohno, Alex C. Snoeren, and David Wetherall. Enlisting ISPs to improve online privacy: IP address mixing by default. In *Proceedings of the 9th Privacy Enhancing Technologies Symposium (PETS)*, 2009.

[70] Franziska Roesner, Tadayoshi Kohno, and David Wetherall. Detecting and defending against third-party tracking on the web. In *Proceedings of the 9th USENIX Symposium on Networked System Design and Implementation (NSDI)*, 2012.

[71] P. Savola. Mtu and fragmentation issues with in-the-network tunneling. RFC 4459, 2006.

[72] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 149–160, 2001.

[73] David Thaler and Chinya V. Ravishankar. Using name-based mappings to increase hit rates. *IEEE/ACM Transactions on Networking*, 6(1):1–14, 1998.

[74] Robert J. Walls, Shane S. Clark, and Brian Neil Levine. Functional privacy or why cookies are better with milk. In *Proceedings of the 7th USENIX Workshop on Hot Topics in Security (HotSec)*, 2012.

[75] Zack Whittaker. Dropbox under fire for 'DMCA takedown' of personal folders, but fears are vastly overblown. `http://www.zdnet.com/dropbox-under-fire-for-dmca-takedown-7000027855`, March 2014.

[76] Craig E. Wills and Can Tatar. Understanding what they do with what they know. In *Proceedings of the 8th ACM Workshop on Privacy in the Electronic Society (WPES)*, 2012.

[77] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V. Madhyastha. SPANStore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 292–308, 2013.

[78] Ting-Fang Yen, Yinglian Xie, Fang Yu, Roger Peng Yu, and Martin Abadi. Host fingerprinting and tracking on the web: Privacy and security implications. In *Proceedings of the 19th Annual Network & Distributed System Security Symposium (NDSS)*, 2012.